

- (*f*asinh *a*)  
 (*f*acosh *a*)   ▷ asinh *a*, acosh *a*, or atanh *a*, respectively.  
 (*f*atanh *a*)
- (*f*cis *a*)           ▷ Return  $e^{i a} = \cos a + i \sin a$ .
- (*f*conjugate *a*)   ▷ Return complex conjugate of *a*.
- (*f*max *num*<sup>+</sup>)  
 (*f*min *num*<sup>+</sup>)   ▷ Greatest or least, respectively, of *nums*.
- $\left( \begin{array}{l} \{ \textit{fround} | \textit{fround} \} \\ \{ \textit{ffloor} | \textit{ffloor} \} \\ \{ \textit{fceiling} | \textit{fceiling} \} \\ \{ \textit{ftruncate} | \textit{ftruncate} \} \end{array} \right) n [d_{\square}]$   
 ▷ Return as **integer** or **float**, respectively,  $n/d$  rounded, or rounded towards  $-\infty$ ,  $+\infty$ , or 0, respectively; and remainder.
- $\left( \begin{array}{l} \{ \textit{fmod} \} \\ \{ \textit{frem} \} \end{array} \right) n d$   
 ▷ Same as *f*floor or *f*truncate, respectively, but return remainder only.
- (*f*random *limit* [*state* random-state\*])  
 ▷ Return non-negative random number less than *limit*, and of the same type.
- (*f*make-random-state [*state* [NIL] T] random)  
 ▷ Copy of **random-state** object *state* or of the current random state; or a randomly initialized fresh random state.
- random-state\*           ▷ Current random state.
- (*f*float-sign *num-a* [*num-b* random])   ▷ num-b with *num-a*'s sign.
- (*f*signum *n*)  
 ▷ Number of magnitude 1 representing sign or phase of *n*.
- (*f*numerator *rational*)  
 (*f*denominator *rational*)  
 ▷ Numerator or denominator, respectively, of *rational*'s canonical form.
- (*f*realpart *number*)  
 (*f*imagpart *number*)  
 ▷ Real part or imaginary part, respectively, of *number*.
- (*f*complex *real* [*imag* random])   ▷ Make a complex number.
- (*f*phase *num*)       ▷ Angle of *num*'s polar representation.
- (*f*abs *n*)           ▷ Return |n|.
- (*f*rational *real*)  
 (*f*rationalize *real*)  
 ▷ Convert *real* to rational. Assume complete/limited accuracy for *real*.
- (*f*float *real* [*prototype* float])  
 ▷ Convert *real* into float with type of *prototype*.

### 1.3 Logic Functions

Negative integers are used in two's complement representation.

- (*f*boole *operation int-a int-b*)  
 ▷ Return value of bitwise logical *operation*. *operations* are
- c*boole-1           ▷ *int-a*.  
*c*boole-2           ▷ *int-b*.  
*c*boole-c1          ▷  $\neg \textit{int-a}$ .  
*c*boole-c2          ▷  $\neg \textit{int-b}$ .  
*c*boole-set         ▷ All bits set.  
*c*boole-clr         ▷ All bits zero.

## Quick Reference

cl

## Common

lisp

Bert Burgemeister

## Contents

<b>1</b>	<b>Numbers</b>	<b>3</b>	9.5	Control Flow . . . . .	19
1.1	Predicates . . . . .	3	9.6	Iteration . . . . .	21
1.2	Numeric Functions . . . . .	3	9.7	Loop Facility . . . . .	21
1.3	Logic Functions . . . . .	4	<b>10</b>	<b>CLOS</b>	<b>23</b>
1.4	Integer Functions . . . . .	5	10.1	Classes . . . . .	23
1.5	Implementation-Dependent . . . . .	6	10.2	Generic Functions . . . . .	25
<b>2</b>	<b>Characters</b>	<b>6</b>	10.3	Method Combination Types . . . . .	26
<b>3</b>	<b>Strings</b>	<b>7</b>	<b>11</b>	<b>Conditions and Errors</b>	<b>27</b>
<b>4</b>	<b>Conses</b>	<b>8</b>	<b>12</b>	<b>Types and Classes</b>	<b>30</b>
4.1	Predicates . . . . .	8	<b>13</b>	<b>Input/Output</b>	<b>32</b>
4.2	Lists . . . . .	8	13.1	Predicates . . . . .	32
4.3	Association Lists . . . . .	9	13.2	Reader . . . . .	32
4.4	Trees . . . . .	10	13.3	Character Syntax . . . . .	33
4.5	Sets . . . . .	10	13.4	Printer . . . . .	34
<b>5</b>	<b>Arrays</b>	<b>10</b>	13.5	Format . . . . .	36
5.1	Predicates . . . . .	10	13.6	Streams . . . . .	39
5.2	Array Functions . . . . .	10	13.7	Pathnames and Files . . . . .	40
5.3	Vector Functions . . . . .	11	<b>14</b>	<b>Packages and Symbols</b>	<b>42</b>
<b>6</b>	<b>Sequences</b>	<b>12</b>	14.1	Predicates . . . . .	42
6.1	Sequence Predicates . . . . .	12	14.2	Packages . . . . .	42
6.2	Sequence Functions . . . . .	12	14.3	Symbols . . . . .	43
<b>7</b>	<b>Hash Tables</b>	<b>14</b>	14.4	Standard Packages . . . . .	44
<b>8</b>	<b>Structures</b>	<b>15</b>	<b>15</b>	<b>Compiler</b>	<b>44</b>
<b>9</b>	<b>Control Structure</b>	<b>15</b>	15.1	Predicates . . . . .	44
9.1	Predicates . . . . .	15	15.2	Compilation . . . . .	44
9.2	Variables . . . . .	16	15.3	REPL and Debugging . . . . .	45
9.3	Functions . . . . .	17	15.4	Declarations . . . . .	46
9.4	Macros . . . . .	18	<b>16</b>	<b>External Environment</b>	<b>47</b>

## Typographic Conventions

**name**; *f***name**; *g***name**; *m***name**; *s***name**; *v*\***name**\*; *c***name**  
 ▷ Symbol defined in Common Lisp; esp. function, generic function, macro, special operator, variable, constant.

*them* ▷ Placeholder for actual code.

*me* ▷ Literal text.

[*foo*bar] ▷ Either one *foo* or nothing; defaults to *bar*.

*foo*\*; {*foo*}\* ▷ Zero or more *foos*.

*foo*<sup>+</sup>; {*foo*}<sup>+</sup> ▷ One or more *foos*.

*foos* ▷ English plural denotes a list argument.

{*foo*|*bar*|*baz*}; {*foo*  
*bar*  
*baz*} ▷ Either *foo*, or *bar*, or *baz*.

{*foo*  
*bar*  
*baz*} ▷ Anything from none to each of *foo*, *bar*, and *baz*.

*foo* ▷ Argument *foo* is not evaluated.

*bar* ▷ Argument *bar* is possibly modified.

*foo*<sup>P<sub>k</sub></sup> ▷ *foo*\* is evaluated as in *sprogn*; see page 20.

*foo*; *bar*; *baz*<sub>2</sub>; *baz*<sub>n</sub> ▷ Primary, secondary, and *n*th return value.

T; NIL ▷ **t**, or truth in general; and **nil** or **()**.

## 1 Numbers

### 1.1 Predicates

(*f* = *number*<sup>+</sup>)  
 (*f* / = *number*<sup>+</sup>)  
 ▷ T if all *numbers*, or none, respectively, are equal in value.

(*f* > *number*<sup>+</sup>)  
 (*f* > = *number*<sup>+</sup>)  
 (*f* < *number*<sup>+</sup>)  
 (*f* < = *number*<sup>+</sup>)  
 ▷ Return T if *numbers* are monotonically decreasing, monotonically non-increasing, monotonically increasing, or monotonically non-decreasing, respectively.

(*f* **minusp** *a*)  
 (*f* **zerop** *a*)  
 (*f* **plusp** *a*)  
 ▷ T if *a* < 0, *a* = 0, or *a* > 0, respectively.

(*f* **evenp** *int*)  
 (*f* **oddp** *int*)  
 ▷ T if *int* is even or odd, respectively.

(*f* **numberp** *foo*)  
 (*f* **realp** *foo*)  
 (*f* **rationalp** *foo*)  
 (*f* **floatp** *foo*)  
 (*f* **integerp** *foo*)  
 (*f* **complexp** *foo*)  
 (*f* **random-state-p** *foo*)  
 ▷ T if *foo* is of indicated type.

### 1.2 Numeric Functions

(*f* + *a*<sub>□</sub><sup>\*</sup>)  
 (*f* \* *a*<sub>□</sub><sup>\*</sup>)  
 ▷ Return  $\sum a$  or  $\prod a$ , respectively.

(*f* - *a* *b*<sup>\*</sup>)  
 (*f* / *a* *b*<sup>\*</sup>)  
 ▷ Return  $a - \sum b$  or  $a / \prod b$ , respectively. Without any *bs*, return  $-a$  or  $1/a$ , respectively.

(*f* **1+** *a*)  
 (*f* **1-** *a*)  
 ▷ Return  $a + 1$  or  $a - 1$ , respectively.

{*m***incf**}  
{*m***decf**}  
*place* [*delta*<sub>□</sub>] )  
 ▷ Increment or decrement the value of *place* by *delta*. Return new value.

(*f* **exp** *p*)  
 (*f* **expt** *b* *p*)  
 ▷ Return  $e^p$  or  $b^p$ , respectively.

(*f* **log** *a* [*b*<sub>□</sub>])  
 ▷ Return  $\log_b a$  or, without *b*,  $\ln a$ .

(*f* **sqrt** *n*)  
 (*f* **isqrt** *n*)  
 ▷  $\sqrt{n}$  in complex numbers/natural numbers.

(*f* **lcm** *integer*<sup>\*</sup> <sub>□</sub>)  
 (*f* **gcd** *integer*<sup>\*</sup> <sub>□</sub>)  
 ▷ Least common multiple or greatest common denominator, respectively, of *integers*. (**gcd**) returns 0.

**pi** ▷ **long-float** approximation of  $\pi$ , Ludolph's number.

(*f* **sin** *a*)  
 (*f* **cos** *a*)  
 (*f* **tan** *a*)  
 ▷ sin *a*, cos *a*, or tan *a*, respectively. (*a* in radians.)

(*f* **asin** *a*)  
 (*f* **acos** *a*)  
 ▷ arcsin *a* or arccos *a*, respectively, in radians.

(*f* **atan** *a* [*b*<sub>□</sub>])  
 ▷ arctan  $\frac{a}{b}$  in radians.

(*f* **sinh** *a*)  
 (*f* **cosh** *a*)  
 (*f* **tanh** *a*)  
 ▷ sinh *a*, cosh *a*, or tanh *a*, respectively.

(*fchar* string *i*)  
 (*fschar* string *i*)  
 ▷ Return zero-indexed *ith* character of string ignoring/obeying, respectively, fill pointer. **setfable**.

(*fparse-integer* string  $\left\{ \begin{array}{l} \text{:start } \text{start}_{\underline{0}} \\ \text{:end } \text{end}_{\underline{NIL}} \\ \text{:radix } \text{int}_{\underline{10}} \\ \text{:junk-allowed } \text{bool}_{\underline{NIL}} \end{array} \right\}$ )  
 ▷ Return integer parsed from *string* and index of parse end.

## 4 Conses

### 4.1 Predicates

(*fconsp* *foo*)  
 (*flistp* *foo*)  
 ▷ Return T if *foo* is of indicated type.

(*fendp* *list*)  
 (*fnull* *foo*)  
 ▷ Return T if *list/foo* is NIL.

(*fatom* *foo*)  
 ▷ Return T if *foo* is not a **cons**.

(*ftailp* *foo list*)  
 ▷ Return T if *foo* is a tail of *list*.

(*fmember* *foo list*  $\left\{ \begin{array}{l} \text{:test } \text{function}_{\underline{\neq}} \\ \text{:test-not } \text{function} \\ \text{:key } \text{function} \end{array} \right\}$ )  
 ▷ Return tail of *list* starting with its first element matching *foo*. Return NIL if there is no such element.

$\left\{ \begin{array}{l} \text{:fmember-if} \\ \text{:fmember-if-not} \end{array} \right\}$  *test list*  $\text{:key } \text{function}$ )  
 ▷ Return tail of *list* starting with its first element satisfying *test*. Return NIL if there is no such element.

(*fsubsetp* *list-a list-b*  $\left\{ \begin{array}{l} \text{:test } \text{function}_{\underline{\neq}} \\ \text{:test-not } \text{function} \\ \text{:key } \text{function} \end{array} \right\}$ )  
 ▷ Return T if *list-a* is a subset of *list-b*.

### 4.2 Lists

(*fcons* *foo bar*)  
 ▷ Return new cons (*foo . bar*).

(*flist* *foo\**)  
 ▷ Return list of *foos*.

(*flist\** *foo+*)  
 ▷ Return list of *foos* with last *foo* becoming cdr of last cons. Return foo if only one *foo* given.

(*fmake-list* *num*  $\text{:initial-element } \text{foo}_{\underline{NIL}}$ )  
 ▷ New list with *num* elements set to *foo*.

(*flist-length* *list*)  
 ▷ Length of *list*; NIL for circular *list*.

(*fcar* *list*)  
 ▷ Car of *list* or NIL if *list* is NIL. **setfable**.

(*fcdr* *list*)  
 (*frest* *list*)  
 ▷ Cdr of *list* or NIL if *list* is NIL. **setfable**.

(*fnthcdr* *n list*)  
 ▷ Return tail of *list* after calling *fcar* *n* times.

$\left\{ \text{:ffirst} \mid \text{:fsecond} \mid \text{:fthird} \mid \text{:fourth} \mid \text{:fifth} \mid \text{:fsixth} \mid \dots \mid \text{:fninth} \mid \text{:ftenth} \right\}$  *list*)  
 ▷ Return nth element of *list* if any, or NIL otherwise. **setfable**.

(*f\_nth* *n list*)  
 ▷ Zero-indexed nth element of *list*. **setfable**.

(*fcarX* *list*)  
 ▷ With *X* being one to four **as** and **ds** representing *fcars* and *fcdrs*, e.g. (*fcar* *bar*) is equivalent to (*fcar* (*fcar* *bar*)). **setfable**.

(*flast* *list* [*num*])  
 ▷ Return list of last *num* conses of *list*.

*cboole-eqv* ▷ *int-a*  $\equiv$  *int-b*.

*cboole-and* ▷ *int-a*  $\wedge$  *int-b*.

*cboole-andc1* ▷  $\neg$ *int-a*  $\wedge$  *int-b*.

*cboole-andc2* ▷ *int-a*  $\wedge$   $\neg$ *int-b*.

*cboole-nand* ▷  $\neg$ (*int-a*  $\wedge$  *int-b*).

*cboole-ior* ▷ *int-a*  $\vee$  *int-b*.

*cboole-orc1* ▷  $\neg$ *int-a*  $\vee$  *int-b*.

*cboole-orc2* ▷ *int-a*  $\vee$   $\neg$ *int-b*.

*cboole-xor* ▷  $\neg$ (*int-a*  $\equiv$  *int-b*).

*cboole-nor* ▷  $\neg$ (*int-a*  $\vee$  *int-b*).

(*flognot* *integer*)  
 ▷  $\neg$ *integer*.

(*flogeqv* *integer\**)  
 (*flogand* *integer\**)  
 ▷ Return value of exclusive-nored or anded *integers*, respectively. Without any *integer*, return  $\neg$ 1.

(*flogandc1* *int-a int-b*)  
 ▷  $\neg$ *int-a*  $\wedge$  *int-b*.

(*flogandc2* *int-a int-b*)  
 ▷ *int-a*  $\wedge$   $\neg$ *int-b*.

(*flognand* *int-a int-b*)  
 ▷  $\neg$ (*int-a*  $\wedge$  *int-b*).

(*flogxor* *integer\**)  
 (*flogior* *integer\**)  
 ▷ Return value of exclusive-ored or ored *integers*, respectively. Without any *integer*, return 0.

(*flogorc1* *int-a int-b*)  
 ▷  $\neg$ *int-a*  $\vee$  *int-b*.

(*flogorc2* *int-a int-b*)  
 ▷ *int-a*  $\vee$   $\neg$ *int-b*.

(*flognor* *int-a int-b*)  
 ▷  $\neg$ (*int-a*  $\vee$  *int-b*).

(*flogbitp* *i int*)  
 ▷ T if zero-indexed *ith* bit of *int* is set.

(*flogtest* *int-a int-b*)  
 ▷ Return T if there is any bit set in *int-a* which is set in *int-b* as well.

(*flogcount* *int*)  
 ▷ Number of 1 bits in *int*  $\geq$  0, number of 0 bits in *int*  $<$  0.

### 1.4 Integer Functions

(*finteger-length* *integer*)  
 ▷ Number of bits necessary to represent *integer*.

(*fldb-test* *byte-spec integer*)  
 ▷ Return T if any bit specified by *byte-spec* in *integer* is set.

(*fash* *integer count*)  
 ▷ Return copy of *integer* arithmetically shifted left by *count* adding zeros at the right, or, for *count*  $<$  0, shifted right discarding bits.

(*fldb* *byte-spec integer*)  
 ▷ Extract byte denoted by *byte-spec* from *integer*. **setfable**.

$\left\{ \begin{array}{l} \text{:fdeposit-field} \\ \text{:fdpb} \end{array} \right\}$  *int-a byte-spec int-b*)  
 ▷ Return *int-b* with bits denoted by *byte-spec* replaced by corresponding bits of *int-a*, or by the low (*fbyte-size* *byte-spec*) bits of *int-a*, respectively.

(*fmask-field* *byte-spec integer*)  
 ▷ Return copy of *integer* with all bits unset but those denoted by *byte-spec*. **setfable**.

(*fbyte* *size position*)  
 ▷ Byte specifier for a byte of *size* bits starting at a weight of  $2^{\text{position}}$ .

(*fbyte-size* *byte-spec*)  
 (*fbyte-position* *byte-spec*)  
 ▷ Size or position, respectively, of *byte-spec*.

## 1.5 Implementation-Dependent

$\left. \begin{array}{l} \text{cshort-float} \\ \text{csingle-float} \\ \text{cdouble-float} \\ \text{clong-float} \end{array} \right\} \begin{array}{l} \text{epsilon} \\ \text{negative-epsilon} \end{array}$

▷ Smallest possible number making a difference when added or subtracted, respectively.

$\left. \begin{array}{l} \text{cleast-negative} \\ \text{cleast-negative-normalized} \\ \text{cleast-positive} \\ \text{cleast-positive-normalized} \end{array} \right\} \begin{array}{l} \text{short-float} \\ \text{single-float} \\ \text{double-float} \\ \text{long-float} \end{array}$

▷ Available numbers closest to  $-0$  or  $+0$ , respectively.

$\left. \begin{array}{l} \text{cmost-negative} \\ \text{cmost-positive} \end{array} \right\} \begin{array}{l} \text{short-float} \\ \text{single-float} \\ \text{double-float} \\ \text{long-float} \\ \text{fixnum} \end{array}$

▷ Available numbers closest to  $-\infty$  or  $+\infty$ , respectively.

(*f*decode-float *n*)

(*f*integer-decode-float *n*)

▷ Return significant, exponent, and sign of float *n*.

(*f*scale-float *n* *i*) ▷ With *n*'s radix *b*, return  $nb^i$ .

(*f*float-radix *n*)

(*f*float-digits *n*)

(*f*float-precision *n*)

▷ Radix, number of digits in that radix, or precision in that radix, respectively, of float *n*.

(*f*upgraded-complex-part-type *foo* [*environment*<sub>☐</sub>])

▷ Type of most specialized **complex** number able to hold parts of type *foo*.

## 2 Characters

The **standard-char** type comprises a-z, A-Z, 0-9, Newline, Space, and !?@"' . : ; \* + - / \ ~ \_ ^ < > # % & ( ) [ ] { }.

(*f*characterp *foo*)

(*f*standard-char-p *char*) ▷ T if argument is of indicated type.

(*f*graphic-char-p *character*)

(*f*alpha-char-p *character*)

(*f*alphanumericp *character*)

▷ T if *character* is visible, alphabetic, or alphanumeric, respectively.

(*f*upper-case-p *character*)

(*f*lower-case-p *character*)

(*f*both-case-p *character*)

▷ Return T if *character* is uppercase, lowercase, or able to be in another case, respectively.

(*f*digit-char-p *character* [*radix*<sub>☐</sub>])

▷ Return its weight if *character* is a digit, or NIL otherwise.

(*f*char= *character*<sup>+</sup>)

(*f*char/= *character*<sup>+</sup>)

▷ Return T if all *characters*, or none, respectively, are equal.

(*f*char-equal *character*<sup>+</sup>)

(*f*char-not-equal *character*<sup>+</sup>)

▷ Return T if all *characters*, or none, respectively, are equal ignoring case.

(*f*char> *character*<sup>+</sup>)

(*f*char>= *character*<sup>+</sup>)

(*f*char< *character*<sup>+</sup>)

(*f*char<= *character*<sup>+</sup>)

▷ Return T if *characters* are monotonically decreasing, monotonically non-increasing, monotonically increasing, or monotonically non-decreasing, respectively.

(*f*char-greaterp *character*<sup>+</sup>)

(*f*char-not-lessp *character*<sup>+</sup>)

(*f*char-lessp *character*<sup>+</sup>)

(*f*char-not-greaterp *character*<sup>+</sup>)

▷ Return T if *characters* are monotonically decreasing, monotonically non-increasing, monotonically increasing, or monotonically non-decreasing, respectively, ignoring case.

(*f*char-upcase *character*)

(*f*char-downcase *character*)

▷ Return corresponding uppercase/lowercase character, respectively.

(*f*digit-char *i* [*radix*<sub>☐</sub>]) ▷ Character representing digit *i*.

(*f*char-name *char*)

▷ *char*'s name if any, or NIL.

(*f*name-char *foo*)

▷ Character named *foo* if any, or NIL.

(*f*char-int *character*)

(*f*char-code *character*)

▷ Code of *character*.

(*f*code-char *code*)

▷ Character with *code*.

*c*char-code-limit ▷ Upper bound of (*f*char-code *char*);  $\geq 96$ .

(*f*character *c*) ▷ Return #\c.

## 3 Strings

Strings can as well be manipulated by array and sequence functions; see pages 10 and 12.

(*f*stringp *foo*)

(*f*simple-string-p *foo*) ▷ T if *foo* is of indicated type.

$\left( \begin{array}{l} \text{fstring=} \\ \text{fstring-equal} \end{array} \right) \text{foo bar} \left\{ \begin{array}{l} \text{:start1 } \text{start-foo}_{\text{☐}} \\ \text{:start2 } \text{start-bar}_{\text{☐}} \\ \text{:end1 } \text{end-foo}_{\text{☐}} \\ \text{:end2 } \text{end-bar}_{\text{☐}} \end{array} \right\}$

▷ Return T if subsequences of *foo* and *bar* are equal. Obey/ignore, respectively, case.

$\left( \begin{array}{l} \text{fstring}\{/= \text{|-not-equal}\} \\ \text{fstring}\{> \text{|-greaterp}\} \\ \text{fstring}\{>= \text{|-not-lessp}\} \\ \text{fstring}\{< \text{|-lessp}\} \\ \text{fstring}\{<= \text{|-not-greaterp}\} \end{array} \right) \text{foo bar} \left\{ \begin{array}{l} \text{:start1 } \text{start-foo}_{\text{☐}} \\ \text{:start2 } \text{start-bar}_{\text{☐}} \\ \text{:end1 } \text{end-foo}_{\text{☐}} \\ \text{:end2 } \text{end-bar}_{\text{☐}} \end{array} \right\}$

▷ If *foo* is lexicographically not equal, greater, not less, less, or not greater, respectively, then return position of first mismatching character in *foo*. Otherwise return NIL. Obey/ignore, respectively, case.

$(\text{fmake-string } \text{size} \left\{ \begin{array}{l} \text{:initial-element } \text{char} \\ \text{:element-type } \text{type}_{\text{character}} \end{array} \right\})$

▷ Return string of length *size*.

(*f*string *x*)

$\left( \begin{array}{l} \text{fstring-capitalize} \\ \text{fstring-upcase} \\ \text{fstring-downcase} \end{array} \right) x \left\{ \begin{array}{l} \text{:start } \text{start}_{\text{☐}} \\ \text{:end } \text{end}_{\text{☐}} \end{array} \right\}$

▷ Convert *x* (**symbol**, **string**, or **character**) into a string, a string with capitalized words, an all-uppercase string, or an all-lowercase string, respectively.

$\left( \begin{array}{l} \text{fnstring-capitalize} \\ \text{fnstring-upcase} \\ \text{fnstring-downcase} \end{array} \right) \widetilde{\text{string}} \left\{ \begin{array}{l} \text{:start } \text{start}_{\text{☐}} \\ \text{:end } \text{end}_{\text{☐}} \end{array} \right\}$

▷ Convert *string* into a string with capitalized words, an all-uppercase string, or an all-lowercase string, respectively.

$\left( \begin{array}{l} \text{fstring-trim} \\ \text{fstring-left-trim} \\ \text{fstring-right-trim} \end{array} \right) \text{char-bag } \text{string}$

▷ Return string with all characters in sequence *char-bag* removed from both ends, from the beginning, or from the end, respectively.

## 6 Sequences

### 6.1 Sequence Predicates

$\left\{ \begin{array}{l} \text{every} \\ \text{notevery} \end{array} \right\} \text{ test sequence}^+$

▷ Return NIL or T, respectively, as soon as *test* on any set of corresponding elements of *sequences* returns NIL.

$\left\{ \begin{array}{l} \text{some} \\ \text{notany} \end{array} \right\} \text{ test sequence}^+$

▷ Return value of *test* or NIL, respectively, as soon as *test* on any set of corresponding elements of *sequences* returns non-NIL.

$(\text{mismatch } \text{sequence-a } \text{sequence-b } \left\{ \begin{array}{l} \text{:from-end } \text{bool} \text{NIL} \\ \text{:test } \text{function} \text{NIL} \\ \text{:test-not } \text{function} \\ \text{:start1 } \text{start-a} \text{0} \\ \text{:start2 } \text{start-b} \text{0} \\ \text{:end1 } \text{end-a} \text{NIL} \\ \text{:end2 } \text{end-b} \text{NIL} \\ \text{:key } \text{function} \end{array} \right\})$

▷ Return position in sequence-a where *sequence-a* and *sequence-b* begin to mismatch. Return NIL if they match entirely.

### 6.2 Sequence Functions

$(\text{make-sequence } \text{sequence-type } \text{size } [\text{:initial-element } \text{foo}])$

▷ Make sequence of *sequence-type* with *size* elements.

$(\text{concatenate } \text{type } \text{sequence}^*)$

▷ Return concatenated sequence of *type*.

$(\text{merge } \text{type } \text{sequence-a } \text{sequence-b } \text{test } [\text{:key } \text{function} \text{NIL}])$

▷ Return interleaved sequence of *type*. Merged sequence will be sorted if both *sequence-a* and *sequence-b* are sorted.

$(\text{fill } \text{sequence } \text{foo } \left\{ \begin{array}{l} \text{:start } \text{start} \text{0} \\ \text{:end } \text{end} \text{NIL} \end{array} \right\})$

▷ Return sequence after setting elements between *start* and *end* to *foo*.

$(\text{length } \text{sequence})$

▷ Return length of sequence (being value of fill pointer if applicable).

$(\text{count } \text{foo } \text{sequence } \left\{ \begin{array}{l} \text{:from-end } \text{bool} \text{NIL} \\ \text{:test } \text{function} \text{NIL} \\ \text{:test-not } \text{function} \\ \text{:start } \text{start} \text{0} \\ \text{:end } \text{end} \text{NIL} \\ \text{:key } \text{function} \end{array} \right\})$

▷ Return number of elements in *sequence* which match *foo*.

$\left\{ \begin{array}{l} \text{count-if} \\ \text{count-if-not} \end{array} \right\} \text{ test } \text{sequence } \left\{ \begin{array}{l} \text{:from-end } \text{bool} \text{NIL} \\ \text{:start } \text{start} \text{0} \\ \text{:end } \text{end} \text{NIL} \\ \text{:key } \text{function} \end{array} \right\})$

▷ Return number of elements in *sequence* which satisfy *test*.

$(\text{elt } \text{sequence } \text{index})$

▷ Return element of sequence pointed to by zero-indexed *index*. **setfable**.

$(\text{subseq } \text{sequence } \text{start } [\text{end} \text{NIL}])$

▷ Return subsequence of sequence between *start* and *end*. **setfable**.

$\left\{ \begin{array}{l} \text{sort} \\ \text{stable-sort} \end{array} \right\} \text{ sequence } \text{test } [\text{:key } \text{function}]$

▷ Return sequence sorted. Order of elements considered equal is not guaranteed/retained, respectively.

$(\text{reverse } \text{sequence})$

$(\text{nreverse } \text{sequence})$

▷ Return sequence in reverse order.

$\left\{ \begin{array}{l} \text{butlast } \text{list} \\ \text{nbutlast } \text{list} \end{array} \right\} [\text{num} \text{NIL}]$  ▷ list excluding last *num* conses.

$\left\{ \begin{array}{l} \text{rplaca} \\ \text{rplacd} \end{array} \right\} \widetilde{\text{cons}} \text{ object}$

▷ Replace *car*, or *cdr*, respectively, of cons with *object*.

$(\text{ldiff } \text{list } \text{foo})$

▷ If *foo* is a tail of *list*, return preceding part of list. Otherwise return list.

$(\text{adjoin } \text{foo } \text{list } \left\{ \begin{array}{l} \text{:test } \text{function} \text{NIL} \\ \text{:test-not } \text{function} \\ \text{:key } \text{function} \end{array} \right\})$

▷ Return list if *foo* is already member of *list*. If not, return (fcons foo list).

$(\text{mpop } \widetilde{\text{place}})$

▷ Set *place* to (fcar place), return (fcdr place).

$(\text{mpush } \text{foo } \widetilde{\text{place}})$  ▷ Set *place* to (fcons foo place).

$(\text{mpushnew } \text{foo } \widetilde{\text{place}} \left\{ \begin{array}{l} \text{:test } \text{function} \text{NIL} \\ \text{:test-not } \text{function} \\ \text{:key } \text{function} \end{array} \right\})$

▷ Set *place* to (fadjoin foo place).

$(\text{append } [\text{proper-list}^* \text{foo} \text{NIL}])$

$(\text{nconc } [\text{non-circular-list}^* \text{foo} \text{NIL}])$

▷ Return concatenated list or, with only one argument, foo. *foo* can be of any type.

$(\text{revappend } \text{list } \text{foo})$

$(\text{nreconc } \text{list } \text{foo})$

▷ Return concatenated list after reversing order in *list*.

$\left\{ \begin{array}{l} \text{mapcar} \\ \text{maplist} \end{array} \right\} \text{ function } \text{list}^+$

▷ Return list of return values of *function* successively invoked with corresponding arguments, either *cars* or *cdrs*, respectively, from each *list*.

$\left\{ \begin{array}{l} \text{mapcan} \\ \text{mapcon} \end{array} \right\} \text{ function } \widetilde{\text{list}}^+$

▷ Return list of concatenated return values of *function* successively invoked with corresponding arguments, either *cars* or *cdrs*, respectively, from each *list*. *function* should return a list.

$\left\{ \begin{array}{l} \text{mapc} \\ \text{mapl} \end{array} \right\} \text{ function } \text{list}^+$

▷ Return first list after successively applying *function* to corresponding arguments, either *cars* or *cdrs*, respectively, from each *list*. *function* should have some side effects.

$(\text{copy-list } \text{list})$  ▷ Return copy of *list* with shared elements.

### 4.3 Association Lists

$(\text{pairlis } \text{keys } \text{values } [\text{alist} \text{NIL}])$

▷ Prepend to alist an association list made from lists *keys* and *values*.

$(\text{acons } \text{key } \text{value } \text{alist})$

▷ Return alist with a (*key* . *value*) pair added.

$\left\{ \begin{array}{l} \text{assoc} \\ \text{rassoc} \end{array} \right\} \text{ foo } \text{alist } \left\{ \begin{array}{l} \text{:test } \text{test} \text{NIL} \\ \text{:test-not } \text{test} \\ \text{:key } \text{function} \end{array} \right\})$

$\left\{ \begin{array}{l} \text{assoc-if}[-\text{not}] \\ \text{rassoc-if}[-\text{not}] \end{array} \right\} \text{ test } \text{alist } [\text{:key } \text{function}]$

▷ First cons whose *car*, or *cdr*, respectively, satisfies *test*.

$(\text{copy-alist } \text{alist})$  ▷ Return copy of *alist*.



## 4.4 Trees

(*f*tree-equal *foo bar* {*:test test* *test* *:test-not test*})

▷ Return **T** if trees *foo* and *bar* have same shape and leaves satisfying *test*.

{*f*subst *new old tree* } {*f*nsubst *new old tree* } {*:test function* *test* *:test-not function* *key function* }

▷ Make copy of *tree* with each subtree or leaf matching *old* replaced by *new*.

{*f*subst-if[-not] *new test tree* } [*:key function*]

▷ Make copy of *tree* with each subtree or leaf satisfying *test* replaced by *new*.

{*f*sublis *association-list tree* } {*f*nsublis *association-list tree* } {*:test function* *test* *:test-not function* *key function* }

▷ Make copy of *tree* with each subtree or leaf matching a key in *association-list* replaced by that key's value.

(*f*copy-tree *tree*) ▷ Copy of *tree* with same shape and leaves.

## 4.5 Sets

{*f*intersection *f*set-difference *f*union *f*set-exclusive-or *f*nintersection *f*nset-difference *f*nunion *f*nset-exclusive-or } *a b* } {*:test function* *test* *:test-not function* *key function* }

▷ Return  $a \cap b$ ,  $a \setminus b$ ,  $a \cup b$ , or  $a \Delta b$ , respectively, of lists *a* and *b*.

## 5 Arrays

### 5.1 Predicates

(*f*arrayp *foo*)

(*f*vectorp *foo*)

(*f*simple-vector-p *foo*) ▷ **T** if *foo* is of indicated type.

(*f*bit-vector-p *foo*)

(*f*simple-bit-vector-p *foo*)

(*f*adjustable-array-p *array*)

(*f*array-has-fill-pointer-p *array*)

▷ **T** if *array* is adjustable/has a fill pointer, respectively.

(*f*array-in-bounds-p *array* [*subscripts*])

▷ Return **T** if *subscripts* are in *array*'s bounds.

### 5.2 Array Functions

{*f*make-array *dimension-sizes* [*:adjustable* *bool* *nil*]}  
{*f*adjust-array *array* *dimension-sizes* }

{*:element-type* *type* *nil*  
*:fill-pointer* {*num* *bool*} *nil*  
*:initial-element* *obj*  
*:initial-contents* *tree-or-array*  
*:displaced-to* *array* [*nil*] [*:displaced-index-offset* *i* *0*]} }

▷ Return fresh, or readjust, respectively, vector or array.

(*f*aref *array* [*subscripts*])

▷ Return array element pointed to by *subscripts*. **setfable**.

(*f*row-major-aref *array* *i*)

▷ Return *i*th element of *array* in row-major order. **setfable**.

(*f*array-row-major-index *array* [*subscripts*])

▷ Index in row-major order of the element denoted by *subscripts*.

(*f*array-dimensions *array*)

▷ List containing the lengths of *array*'s dimensions.

(*f*array-dimension *array* *i*)

▷ Length of *i*th dimension of *array*.

(*f*array-total-size *array*)

▷ Number of elements in *array*.

(*f*array-rank *array*)

▷ Number of dimensions of *array*.

(*f*array-displacement *array*)

▷ Target array and offset.

(*f*bit *bit-array* [*subscripts*])

(*f*sbit *simple-bit-array* [*subscripts*])

▷ Return element of *bit-array* or of *simple-bit-array*. **setfable**.

(*f*bit-not *bit-array* [*result-bit-array* *nil*])

▷ Return result of bitwise negation of *bit-array*. If *result-bit-array* is **T**, put result in *bit-array*; if it is **NIL**, make a new array for result.

{*f*bit-eqv *f*bit-and *f*bit-andc1 *f*bit-andc2 *f*bit-nand *f*bit-ior *f*bit-orc1 *f*bit-orc2 *f*bit-xor *f*bit-nor } *bit-array-a bit-array-b* [*result-bit-array* *nil*])

▷ Return result of bitwise logical operations (cf. operations of **fboole**, page 4) on *bit-array-a* and *bit-array-b*. If *result-bit-array* is **T**, put result in *bit-array-a*; if it is **NIL**, make a new array for result.

*c*array-rank-limit

▷ Upper bound of array rank;  $\geq 8$ .

*c*array-dimension-limit

▷ Upper bound of an array dimension;  $\geq 1024$ .

*c*array-total-size-limit

▷ Upper bound of array size;  $\geq 1024$ .

### 5.3 Vector Functions

Vectors can as well be manipulated by sequence functions; see section 6.

(*f*vector *foo*\*)

▷ Return fresh simple vector of *foos*.

(*f*svref *vector* *i*)

▷ Element *i* of simple *vector*. **setfable**.

(*f*vector-push *foo* *vector*)

▷ Return **NIL** if *vector*'s fill pointer equals size of *vector*. Otherwise replace element of *vector* pointed to by fill pointer with *foo*; then increment fill pointer.

(*f*vector-push-extend *foo* *vector* [*num*])

▷ Replace element of *vector* pointed to by fill pointer with *foo*, then increment fill pointer. Extend *vector*'s size by  $\geq$  *num* if necessary.

(*f*vector-pop *vector*)

▷ Return element of *vector* its fillpointer points to after decrementation.

(*f*fill-pointer *vector*)

▷ Fill pointer of *vector*. **setfable**.

- (*f*functionp *foo*)      ▷ T if *foo* is of type **function**.
- (*f*fboundp  $\left\{ \begin{array}{l} \textit{foo} \\ \textit{(setf foo)} \end{array} \right\}$ )      ▷ T if *foo* is a global function or macro.

## 9.2 Variables

- $\left\{ \begin{array}{l} \textit{(mdefconstant)} \\ \textit{(mdefparameter)} \end{array} \right\} \widehat{foo} \textit{form} [\widehat{doc}]$   
 ▷ Assign value of *form* to global constant/dynamic variable *foo*.
- (*m*defvar  $\widehat{foo}$  [*form* [*doc*]])  
 ▷ Unless bound already, assign value of *form* to dynamic variable *foo*.
- $\left\{ \begin{array}{l} \textit{(msetf)} \\ \textit{(mpsetf)} \end{array} \right\} \{ \textit{place form} \}^*$   
 ▷ Set *places* to primary values of *forms*. Return values of last *form*/NIL; work sequentially/in parallel, respectively.
- $\left\{ \begin{array}{l} \textit{(ssetq)} \\ \textit{(mpsetq)} \end{array} \right\} \{ \textit{symbol form} \}^*$   
 ▷ Set *symbols* to primary values of *forms*. Return value of last *form*/NIL; work sequentially/in parallel, respectively.
- (*f*set  $\widetilde{\textit{symbol}} \textit{foo}$ )      ▷ Set *symbol*'s value cell to *foo*. Deprecated.
- (*m*multiple-value-setq *vars form*)  
 ▷ Set elements of *vars* to the values of *form*. Return *form*'s primary value.
- (*m*shiftf  $\widetilde{\textit{place}}^+ \textit{foo}$ )  
 ▷ Store value of *foo* in rightmost *place* shifting values of *places* left, returning first place.
- (*m*rotatef  $\widetilde{\textit{place}}^*$ )  
 ▷ Rotate values of *places* left, old first becoming new last *place*'s value. Return NIL.
- (*f*makunbound  $\widetilde{\textit{foo}}$ )      ▷ Delete special variable *foo* if any.
- (*f*get *symbol* *key* [*default*/NIL])  
 (*f*getf *place* *key* [*default*/NIL])  
 ▷ First entry *key* from property list stored in *symbol*/in *place*, respectively, or *default* if there is no *key*. **setfable**.
- (*f*get-properties *property-list* *keys*)  
 ▷ Return key and value of first entry from *property-list* matching a key from <sup>2</sup>*keys*, and tail of *property-list* starting with that key. Return NIL, NIL<sub>2</sub>, and NIL<sub>3</sub> if there was no matching key in *property-list*.
- (*f*remprop  $\widetilde{\textit{symbol}} \textit{key}$ )  
 (*m*remf *place* *key*)  
 ▷ Remove first entry *key* from property list stored in *symbol*/in *place*, respectively. Return T if *key* was there, or NIL otherwise.
- (*s*progv *symbols* *values* *form*<sup>P<sub>k</sub></sup>)  
 ▷ Evaluate *forms* with locally established dynamic bindings of *symbols* to *values* or NIL. Return values of forms.
- $\left\{ \begin{array}{l} \textit{(slet)} \\ \textit{(slet*)} \end{array} \right\} \left( \left\{ \begin{array}{l} \textit{name} \\ \textit{(name [value/NIL])} \end{array} \right\}^* \right) (\textit{declare} \widehat{\textit{decl}}^*)^* \textit{form}^{\text{P}_k}$   
 ▷ Evaluate *forms* with *names* lexically bound (in parallel or sequentially, respectively) to *values*. Return values of forms.
- (*m*multiple-value-bind ( $\widetilde{\textit{var}}^*$ ) *values-form* (*declare*  $\widehat{\textit{decl}}^*$ )<sup>P<sub>k</sub></sup> *body-form*<sup>P<sub>k</sub></sup>)  
 ▷ Evaluate *body-forms* with *vars* lexically bound to the return values of *values-form*. Return values of body-forms.

$$\left\{ \begin{array}{l} \textit{(find)} \\ \textit{(fposition)} \end{array} \right\} \textit{foo sequence} \left\{ \begin{array}{l} \textit{:from-end} \textit{bool} \textit{NIL} \\ \textit{:test} \textit{function} \textit{[#'eq]} \\ \textit{:test-not} \textit{test} \\ \textit{:start} \textit{start} \textit{0} \\ \textit{:end} \textit{end} \textit{NIL} \\ \textit{:key} \textit{function} \end{array} \right\}$$

▷ Return first element in *sequence* which matches *foo*, or its position relative to the begin of *sequence*, respectively.

$$\left\{ \begin{array}{l} \textit{(find-if)} \\ \textit{(find-if-not)} \\ \textit{(fposition-if)} \\ \textit{(fposition-if-not)} \end{array} \right\} \textit{test sequence} \left\{ \begin{array}{l} \textit{:from-end} \textit{bool} \textit{NIL} \\ \textit{:start} \textit{start} \textit{0} \\ \textit{:end} \textit{end} \textit{NIL} \\ \textit{:key} \textit{function} \end{array} \right\}$$

▷ Return first element in *sequence* which satisfies *test*, or its position relative to the begin of *sequence*, respectively.

$$\left\{ \begin{array}{l} \textit{(fsearch)} \end{array} \right\} \textit{sequence-a sequence-b} \left\{ \begin{array}{l} \textit{:from-end} \textit{bool} \textit{NIL} \\ \textit{:test} \textit{function} \textit{[#'eq]} \\ \textit{:test-not} \textit{function} \\ \textit{:start1} \textit{start-a} \textit{0} \\ \textit{:start2} \textit{start-b} \textit{0} \\ \textit{:end1} \textit{end-a} \textit{NIL} \\ \textit{:end2} \textit{end-b} \textit{NIL} \\ \textit{:key} \textit{function} \end{array} \right\}$$

▷ Search *sequence-b* for a subsequence matching *sequence-a*. Return position in *sequence-b*, or NIL.

$$\left\{ \begin{array}{l} \textit{(fremove)} \\ \textit{(fdelete)} \end{array} \right\} \textit{foo sequence} \left\{ \begin{array}{l} \textit{:from-end} \textit{bool} \textit{NIL} \\ \textit{:test} \textit{function} \textit{[#'eq]} \\ \textit{:test-not} \textit{function} \\ \textit{:start} \textit{start} \textit{0} \\ \textit{:end} \textit{end} \textit{NIL} \\ \textit{:key} \textit{function} \\ \textit{:count} \textit{count} \textit{NIL} \end{array} \right\}$$

▷ Make copy of sequence without elements matching *foo*.

$$\left\{ \begin{array}{l} \textit{(fremove-if)} \\ \textit{(fremove-if-not)} \\ \textit{(fdelete-if)} \\ \textit{(fdelete-if-not)} \end{array} \right\} \textit{test sequence} \left\{ \begin{array}{l} \textit{:from-end} \textit{bool} \textit{NIL} \\ \textit{:start} \textit{start} \textit{0} \\ \textit{:end} \textit{end} \textit{NIL} \\ \textit{:key} \textit{function} \\ \textit{:count} \textit{count} \textit{NIL} \end{array} \right\}$$

▷ Make copy of sequence with all (or *count*) elements satisfying *test* removed.

$$\left\{ \begin{array}{l} \textit{(fremove-duplicates)} \\ \textit{(fdelete-duplicates)} \end{array} \right\} \textit{sequence} \left\{ \begin{array}{l} \textit{:from-end} \textit{bool} \textit{NIL} \\ \textit{:test} \textit{function} \textit{[#'eq]} \\ \textit{:test-not} \textit{function} \\ \textit{:start} \textit{start} \textit{0} \\ \textit{:end} \textit{end} \textit{NIL} \\ \textit{:key} \textit{function} \end{array} \right\}$$

▷ Make copy of sequence without duplicates.

$$\left\{ \begin{array}{l} \textit{(fsubstitute)} \\ \textit{(fnsubstitute)} \end{array} \right\} \textit{new old sequence} \left\{ \begin{array}{l} \textit{:from-end} \textit{bool} \textit{NIL} \\ \textit{:test} \textit{function} \textit{[#'eq]} \\ \textit{:test-not} \textit{function} \\ \textit{:start} \textit{start} \textit{0} \\ \textit{:end} \textit{end} \textit{NIL} \\ \textit{:key} \textit{function} \\ \textit{:count} \textit{count} \textit{NIL} \end{array} \right\}$$

▷ Make copy of sequence with all (or *count*) *olds* replaced by *new*.

$$\left\{ \begin{array}{l} \textit{(fsubstitute-if)} \\ \textit{(fsubstitute-if-not)} \\ \textit{(fnsubstitute-if)} \\ \textit{(fnsubstitute-if-not)} \end{array} \right\} \textit{new test sequence} \left\{ \begin{array}{l} \textit{:from-end} \textit{bool} \textit{NIL} \\ \textit{:start} \textit{start} \textit{0} \\ \textit{:end} \textit{end} \textit{NIL} \\ \textit{:key} \textit{function} \\ \textit{:count} \textit{count} \textit{NIL} \end{array} \right\}$$

▷ Make copy of sequence with all (or *count*) elements satisfying *test* replaced by *new*.

$$\left\{ \begin{array}{l} \textit{(freplace)} \end{array} \right\} \textit{sequence-a sequence-b} \left\{ \begin{array}{l} \textit{:start1} \textit{start-a} \textit{0} \\ \textit{:start2} \textit{start-b} \textit{0} \\ \textit{:end1} \textit{end-a} \textit{NIL} \\ \textit{:end2} \textit{end-b} \textit{NIL} \end{array} \right\}$$

▷ Replace elements of *sequence-a* with elements of *sequence-b*.

(*f*map *type function sequence*<sup>+</sup>)

▷ Apply *function* successively to corresponding elements of the *sequences*. Return values as a sequence of *type*. If *type* is NIL, return NIL.

(*f*map-into *result-sequence function sequence*<sup>\*</sup>)

▷ Store into *result-sequence* successively values of *function* applied to corresponding elements of the *sequences*.

(*f*reduce *function sequence*  $\left\{ \begin{array}{l} \text{:initial-value } \widehat{foo} \text{NIL} \\ \text{:from-end } \widehat{bool} \text{NIL} \\ \text{:start } \widehat{start} \text{NIL} \\ \text{:end } \widehat{end} \text{NIL} \\ \text{:key } \widehat{function} \end{array} \right\}$ )

▷ Starting with the first two elements of *sequence*, apply *function* successively to its last return value together with the next element of *sequence*. Return last value of function.

(*f*copy-seq *sequence*)

▷ Copy of *sequence* with shared elements.

## 7 Hash Tables

The Loop Facility provides additional hash table-related functionality; see **loop**, page 21.

Key-value storage similar to hash tables can as well be achieved using association lists and property lists; see pages 9 and 16.

(*f*hash-table-p *foo*) ▷ Return T if *foo* is of type **hash-table**.

(*f*make-hash-table  $\left\{ \begin{array}{l} \text{:test } \{ \widehat{f} \text{eq} | \widehat{f} \text{eql} | \widehat{f} \text{equal} | \widehat{f} \text{equalp} \} \text{NIL} \\ \text{:size } \widehat{int} \\ \text{:rehash-size } \widehat{num} \\ \text{:rehash-threshold } \widehat{num} \end{array} \right\}$ )

▷ Make a hash table.

(*f*gethash *key hash-table* [*default* NIL])

▷ Return object with *key* if any or *default* otherwise; and T if found, NIL otherwise. **setfable**.

(*f*hash-table-count *hash-table*)

▷ Number of entries in *hash-table*.

(*f*remhash *key hash-table*)

▷ Remove from *hash-table* entry with *key* and return T if it existed. Return NIL otherwise.

(*f*clrhash *hash-table*) ▷ Empty hash-table.

(*f*maphash *function hash-table*)

▷ Iterate over *hash-table* calling *function* on key and value. Return NIL.

(*m*with-hash-table-iterator (*foo hash-table*) (**declare**  $\widehat{decl}^*$ )<sup>\*</sup> *form*<sup>P<sub>k</sub></sup>)

▷ Return values of forms. In *forms*, invocations of (*foo*) return: T if an entry is returned; its key; its value.

(*f*hash-table-test *hash-table*)

▷ Test function used in *hash-table*.

(*f*hash-table-size *hash-table*)

(*f*hash-table-rehash-size *hash-table*)

(*f*hash-table-rehash-threshold *hash-table*)

▷ Current size, rehash-size, or rehash-threshold, respectively, as used in *f*make-hash-table.

(*f*sxhash *foo*)

▷ Hash code unique for any argument *f*equal *foo*.

## 8 Structures

(*m*defstruct

*foo*  $\left\{ \begin{array}{l} \text{:conc-name} \\ \text{:conc-name } [\widehat{slot-prefix} \text{foo-}] \\ \text{:constructor} \\ \text{:constructor } [\widehat{maker} \text{MAKE-foo} \text{ [(ord-}\lambda^* \text{)]]}] \\ \text{:copier} \\ \text{:copier } [\widehat{copier} \text{COPY-foo}] \\ \text{:include } \widehat{struct} \left\{ \begin{array}{l} \text{slot} \\ \text{(slot } [\widehat{init} \text{ } \left\{ \begin{array}{l} \text{:type } \widehat{sl-type} \\ \text{:read-only } \widehat{b} \end{array} \right\} ]]) \end{array} \right\}^* \\ \text{:type } \left\{ \begin{array}{l} \text{list} \\ \text{vector} \\ \text{(vector } \widehat{type}) \end{array} \right\} \text{ [(initial-offset } \widehat{n})] \\ \text{:print-object } [\widehat{o-printer}] \\ \text{:print-function } [\widehat{f-printer}] \\ \text{:named} \\ \text{:predicate} \\ \text{:predicate } [\widehat{p-name} \text{foo-P}] \end{array} \right\}$

$\left\{ \begin{array}{l} \text{slot} \\ \text{(slot } [\widehat{init} \text{ } \left\{ \begin{array}{l} \text{:type } \widehat{slot-type} \\ \text{:read-only } \widehat{bool} \end{array} \right\} ]]) \end{array} \right\}^*$

$\left\{ \begin{array}{l} \text{slot} \\ \text{(slot } [\widehat{init} \text{ } \left\{ \begin{array}{l} \text{:type } \widehat{slot-type} \\ \text{:read-only } \widehat{bool} \end{array} \right\} ]]) \end{array} \right\}^*$

▷ Define structure *foo* together with functions MAKE-foo, COPY-foo and foo-P; and **setfable** accessors foo-slot. Instances are of class *foo* or, if **defstruct** option **:type** is given, of the specified type. They can be created by (MAKE-foo {*slot value*}<sup>\*</sup>) or, if ord-λ is given, by (maker *arg*<sup>\*</sup> {*key value*}<sup>\*</sup>). In the latter case, *args* and *keys* correspond to the positional and keyword parameters defined in ord-λ whose *vars* in turn correspond to *slots*. **:print-object**/**:print-function** generate a **gprint-object** method for an instance *bar* of *foo* calling (*o-printer bar stream*) or (*f-printer bar stream print-level*), respectively. If **:type** without **:named** is given, no foo-P is created.

(*f*copy-structure *structure*)

▷ Return copy of structure with shared slot values.

## 9 Control Structure

### 9.1 Predicates

(*f*eq *foo bar*) ▷ T if *foo* and *bar* are identical.

(*f*eql *foo bar*)

▷ T if *foo* and *bar* are identical, or the same **character**, or **numbers** of the same type and value.

(*f*equal *foo bar*)

▷ T if *foo* and *bar* are *f*eql, or are equivalent **pathnames**, or are **conses** with *f*equal cars and cdrs, or are **strings** or **bit-vectors** with *f*eql elements below their fill pointers.

(*f*equalp *foo bar*)

▷ T if *foo* and *bar* are identical; or are the same **character** ignoring case; or are **numbers** of the same value ignoring type; or are equivalent **pathnames**; or are **conses** or **arrays** of the same shape with *f*equalp elements; or are structures of the same type with *f*equalp elements; or are **hash-tables** of the same size with the same **:test** function, the same keys in terms of **:test** function, and *f*equalp elements.

(*f*not *foo*)

▷ T if *foo* is NIL; NIL otherwise.

(*f*boundp *symbol*)

▷ T if *symbol* is a special variable.

(*f*constantp *foo* [*environment* NIL])

▷ T if *foo* is a constant form.



(*m*case test ( $\widehat{\text{key}}$ )  $\text{foo}^{\text{P}_k}$ )\* [( $\text{T}$ )  $\widehat{\text{bar}}^{\text{P}_k}$ ]<sub>NIL</sub>])  
 ▷ Return the values of the first *foo*\* one of whose *keys* is **eq** *test*. Return values of bars if there is no matching *key*.

( $\widehat{\text{m}}_{\text{ccase}}$ ) test ( $\widehat{\text{key}}$ )  $\text{foo}^{\text{P}_k}$ )  
 ▷ Return the values of the first *foo*\* one of whose *keys* is **eq** *test*. Signal non-correctable/correctable **type-error** if there is no matching *key*.

(*m*and *form*\*<sub>P</sub>)  
 ▷ Evaluate *forms* from left to right. Immediately return **NIL** if one *form*'s value is **NIL**. Return values of last *form* otherwise.

(*m*or *form*\*<sub>NIL</sub>)  
 ▷ Evaluate *forms* from left to right. Immediately return primary value of first non-NIL-evaluating form, or all values if last *form* is reached. Return **NIL** if no *form* returns **T**.

(*s*progn *form*\*<sub>NIL</sub>)  
 ▷ Evaluate *forms* sequentially. Return values of last *form*.

(*s*multiple-value-prog1 *form-r* *form*\*<sub>r</sub>)

(*m*prog1 *form-r* *form*\*<sub>r</sub>)

(*m*prog2 *form-a* *form-r* *form*\*<sub>r</sub>)

▷ Evaluate forms in order. Return values/primary value, respectively, of *form-r*.

( $\widehat{\text{m}}_{\text{prog}}$ ) ( $\widehat{\text{m}}_{\text{prog}}$ ) ( $\widehat{\text{tag}}$   $\widehat{\text{form}}$ )  
 ▷ Evaluate *s*tagbody-like body with *names* lexically bound (in parallel or sequentially, respectively) to *values*. Return **NIL** or explicitly *m*returned values. Implicitly, the whole form is a *s*block named **NIL**.

(*s*unwind-protect *protected* *cleanup*\*)  
 ▷ Evaluate *protected* and then, no matter how control leaves *protected*, *cleanups*. Return values of *protected*.

(*s*block *name* *form*\*<sub>P</sub>)  
 ▷ Evaluate *forms* in a lexical environment, and return their values unless interrupted by *s*return-from.

(*s*return-from *foo* [*result*]<sub>NIL</sub>)

(*m*return [*result*]<sub>NIL</sub>)

▷ Have nearest enclosing *s*block named *foo*/named **NIL**, respectively, return with values of *result*.

(*s*tagbody { $\widehat{\text{tag}}$ }*form*\*<sub>r</sub>)

▷ Evaluate *forms* in a lexical environment. *tags* (symbols or integers) have lexical scope and dynamic extent, and are targets for *s*go. Return **NIL**.

(*s*go  $\widehat{\text{tag}}$ )

▷ Within the innermost possible enclosing *s*tagbody, jump to a tag *f*eq *tag*.

(*s*catch *tag* *form*\*<sub>P</sub>)

▷ Evaluate *forms* and return their values unless interrupted by *s*throw.

(*s*throw *tag* *form*)

▷ Have the nearest dynamically enclosing *s*catch with a tag *f*eq *tag* return with the values of *form*.

(*f*sleep *n*) ▷ Wait *n* seconds; return **NIL**.

(*m*destructuring-bind *destruct-λ* *bar* (**declare**  $\widehat{\text{decl}}$ )\* *form*\*<sub>P</sub>)

▷ Evaluate *forms* with variables from tree *destruct-λ* bound to corresponding elements of tree *bar*, and return their values. *destruct-λ* resembles *macro-λ* (section 9.4), but without any **&environment** clause.

### 9.3 Functions

Below, ordinary lambda list (*ord-λ*\*) has the form

(*var*\* [**&optional** (*var* [*init*]<sub>NIL</sub> [*supplied-p*])])\* [**&rest** *var*]

[**&key** (*var* (*var* [*init*]<sub>NIL</sub> [*supplied-p*]) [*key* *var*]) [**&allow-other-keys**]]

[**&aux** (*var* [*init*]<sub>NIL</sub>)])\*].

*supplied-p* is **T** if there is a corresponding argument. *init* forms can refer to any *init* and *supplied-p* to their left.

( $\widehat{\text{m}}_{\text{defun}}$  (*foo* (*ord-λ*\*) (**setf** *foo*) (*new-value* *ord-λ*\*) ( $\widehat{\text{decl}}$ )\*<sub>doc</sub>)  
 ( $\widehat{\text{m}}_{\text{lambda}}$  (*ord-λ*\*) *form*\*<sub>P</sub>)

▷ Define a function named *foo* or (**setf** *foo*), or an anonymous function, respectively, which applies *forms* to *ord-λ*s. For *m*defun, *forms* are enclosed in an implicit *s*block named *foo*.

( $\widehat{\text{s}}_{\text{flet}}$ ) ( $\widehat{\text{s}}_{\text{labels}}$ ) ((*foo* (*ord-λ*\*) (**setf** *foo*) (*new-value* *ord-λ*\*) ( $\widehat{\text{decl}}$ )\*<sub>doc</sub>) *local-form*\*<sub>P</sub>) (**declare**  $\widehat{\text{decl}}$ )\*<sub>form\*<sub>P</sub>)</sub>

▷ Evaluate *forms* with locally defined functions *foo*. Globally defined functions of the same name are shadowed. Each *foo* is also the name of an implicit *s*block around its corresponding *local-form*\*. Only for *s*labels, functions *foo* are visible inside *local-forms*. Return values of *forms*.

(*s*function {*foo* ( $\widehat{\text{m}}_{\text{lambda}}$  *form*\*<sub>r</sub>)})

▷ Return lexically innermost function named *foo* or a lexical closure of the *m*lambda expression.

(*f*apply {*function* (**setf** *function*)} *arg*\* *args*)

▷ Values of *function* called with *args* and the list elements of *args*. **setfable** if *function* is one of *f*aref, *f*bit, and *f*sbit.

(*f*funcall *function* *arg*\*) ▷ Values of *function* called with *args*.

(*s*multiple-value-call *function* *form*\*)

▷ Call *function* with all the values of each *form* as its arguments. Return values returned by *function*.

(*f*values-list *list*) ▷ Return elements of *list*.

(*f*values *foo*\*)

▷ Return as multiple values the primary values of the *foos*. **setfable**.

(*f*multiple-value-list *form*) ▷ List of the values of *form*.

(*m*nth-value *n* *form*)

▷ Zero-indexed *n*th return value of *form*.

(*f*complement *function*)

▷ Return new function with same arguments and same side effects as *function*, but with complementary truth value.

(*f*constantly *foo*)

▷ Function of any number of arguments returning *foo*.

(*f*identity *foo*) ▷ Return *foo*.

(*f*function-lambda-expression *function*)  
 ▷ If available, return lambda expression of *function*, NIL if *function* was defined in an environment without bindings, and name of *function*.

(*f*fdefinition  $\left\{ \begin{array}{l} \text{foo} \\ (\text{setf } \text{foo}) \end{array} \right\}$ )

▷ Definition of global function *foo*. **setfable**.

(*f*fmakunbound *foo*)

▷ Remove global function or macro definition *foo*.

*c*call-arguments-limit

*l*ambda-parameters-limit

▷ Upper bound of the number of function arguments or lambda list parameters, respectively;  $\geq 50$ .

*m*ultiple-values-limit

▷ Upper bound of the number of values a multiple value can have;  $\geq 20$ .

## 9.4 Macros

Below, macro lambda list (*macro-λ\**) has the form of either

(**&whole** *var* [*E*]  $\left\{ \begin{array}{l} \text{var} \\ (\text{macro-}\lambda^*) \end{array} \right\}^*$  [*E*])

**&optional**  $\left\{ \begin{array}{l} \text{var} \\ \left( \left\{ \begin{array}{l} \text{var} \\ (\text{macro-}\lambda^*) \end{array} \right\} [\text{init}_{\text{NIL}} [\text{supplied-}p]] \right) \end{array} \right\}^*$  [*E*]

**&rest**  $\left\{ \begin{array}{l} \text{rest-var} \\ (\text{macro-}\lambda^*) \end{array} \right\}$  [*E*]

**&key**  $\left\{ \begin{array}{l} \text{var} \\ \left( \left\{ \begin{array}{l} \text{var} \\ (:key \left\{ \begin{array}{l} \text{var} \\ (\text{macro-}\lambda^*) \end{array} \right\}) \end{array} \right\} [\text{init}_{\text{NIL}} [\text{supplied-}p]] \right) \end{array} \right\}^*$  [*E*]

**&allow-other-keys** [**&aux**  $\left\{ \begin{array}{l} \text{var} \\ (\text{var} [\text{init}_{\text{NIL}}]) \end{array} \right\}^*$  [*E*])

or

(**&whole** *var* [*E*]  $\left\{ \begin{array}{l} \text{var} \\ (\text{macro-}\lambda^*) \end{array} \right\}^*$  [*E*] [**&optional**

$\left\{ \begin{array}{l} \text{var} \\ \left( \left\{ \begin{array}{l} \text{var} \\ (\text{macro-}\lambda^*) \end{array} \right\} [\text{init}_{\text{NIL}} [\text{supplied-}p]] \right) \end{array} \right\}^*$  [*E*] . *rest-var*).

One toplevel [*E*] may be replaced by **&environment** *var*. *supplied-p* is T if there is a corresponding argument. *init* forms can refer to any *init* and *supplied-p* to their left.

$\left\{ \begin{array}{l} \text{mdefmacro} \\ \text{fdefine-compiler-macro} \end{array} \right\} \left\{ \begin{array}{l} \text{foo} \\ (\text{setf } \text{foo}) \end{array} \right\} (\text{macro-}\lambda^*)$   
 $\left\{ \begin{array}{l} (\text{declare } \widehat{\text{decl}}^*) \\ \widehat{\text{doc}} \end{array} \right\} \text{form}^{\text{P}_k}$

▷ Define macro *foo* which on evaluation as (*foo tree*) applies expanded *forms* to arguments from *tree*, which corresponds to *tree-shaped macro-λs*. *forms* are enclosed in an implicit **block** named *foo*.

(*m*define-symbol-macro *foo form*)

▷ Define symbol macro *foo* which on evaluation evaluates expanded *form*.

(*s*macrolet ((*foo* (*macro-λ\**)  $\left\{ \begin{array}{l} (\text{declare } \widehat{\text{local-decl}}^*) \\ \widehat{\text{doc}} \end{array} \right\}$  )

*macro-form*<sup>P<sub>k</sub></sup>) (declare  $\widehat{\text{decl}}^*$ ) *form*<sup>P<sub>k</sub></sup>)

▷ Evaluate *forms* with locally defined mutually invisible macros *foo* which are enclosed in implicit **blocks** of the same name.

(*s*symbol-macrolet ((*foo expansion-form*)<sup>\*</sup>) (declare  $\widehat{\text{decl}}^*$ ) *form*<sup>P<sub>k</sub></sup>)

▷ Evaluate *forms* with locally defined symbol macros *foo*.

(*m*defsetf *function*  $\left\{ \begin{array}{l} \widehat{\text{updater}} [\widehat{\text{doc}}] \\ (\text{setf-}\lambda^*) (s\text{-var}^*) \end{array} \right\} \left\{ \begin{array}{l} (\text{declare } \widehat{\text{decl}}^*) \\ \widehat{\text{doc}} \end{array} \right\} \text{form}^{\text{P}_k}$  )

where defsetf lambda list (*setf-λ\**) has the form

$(\text{var}^* [\text{\&optional} \left\{ \begin{array}{l} \text{var} \\ (\text{var} [\text{init}_{\text{NIL}} [\text{supplied-}p]]) \end{array} \right\}^* ] [\text{\&rest } \text{var}]$   
 $[\text{\&key} \left\{ \begin{array}{l} \text{var} \\ \left( \left\{ \begin{array}{l} \text{var} \\ (:key \text{ var}) \end{array} \right\} [\text{init}_{\text{NIL}} [\text{supplied-}p]] \right) \end{array} \right\}^* ]$

**&allow-other-keys**] [**&environment** *var*])

▷ Specify how to **setf** a place accessed by *function*.  
**Short form:** (**setf** (*function arg*<sup>\*</sup>) *value-form*) is replaced by (*updater arg*<sup>\*</sup> *value-form*); the latter must return *value-form*.  
**Long form:** on invocation of (**setf** (*function arg*<sup>\*</sup>) *value-form*), *forms* must expand into code that sets the place accessed where *setf-λ* and *s-var*<sup>\*</sup> describe the arguments of *function* and the value(s) to be stored, respectively; and that returns the value(s) of *s-var*<sup>\*</sup>. *forms* are enclosed in an implicit **block** named *function*.

(*m*define-setf-expander *function* (*macro-λ\**)  $\left\{ \begin{array}{l} (\text{declare } \widehat{\text{decl}}^*) \\ \widehat{\text{doc}} \end{array} \right\}$

*form*<sup>P<sub>k</sub></sup>)

▷ Specify how to **setf** a place accessed by *function*. On invocation of (**setf** (*function arg*<sup>\*</sup>) *value-form*), *form*<sup>\*</sup> must expand into code returning *arg-vars*, *args*, *newval-vars*, *set-form*, and *get-form* as described with *f*get-setf-expansion where the elements of macro lambda list *macro-λ\** are bound to corresponding *args*. *forms* are enclosed in an implicit **block** named *function*.

(*f*get-setf-expansion *place* [*environment*<sub>NIL</sub>])

▷ Return lists of temporary variables *arg-vars* and of corresponding *args* as given with *place*, list *newval-vars* with temporary variables corresponding to the new values, and *set-form* and *get-form* specifying in terms of *arg-vars* and *newval-vars* how to **setf** and how to read *place*.

(*m*define-modify-macro *foo* ([**&optional**

$\left\{ \begin{array}{l} \text{var} \\ (\text{var} [\text{init}_{\text{NIL}} [\text{supplied-}p]]) \end{array} \right\}^*$  [**&rest** *var*] *function* [ $\widehat{\text{doc}}$ ])

▷ Define macro *foo* able to modify a place. On invocation of (*foo place arg*<sup>\*</sup>), the value of *function* applied to *place* and *args* will be stored into *place* and returned.

*l*ambda-list-keywords

▷ List of macro lambda list keywords. These are at least:

**&whole** *var*

▷ Bind *var* to the entire macro call form.

**&optional** *var*<sup>\*</sup>

▷ Bind *vars* to corresponding arguments if any.

**&rest|&body** *var*

▷ Bind *var* to a list of remaining arguments.

**&key** *var*<sup>\*</sup>

▷ Bind *vars* to corresponding keyword arguments.

**&allow-other-keys**

▷ Suppress keyword argument checking. Callers can do so using **:allow-other-keys** T.

**&environment** *var*

▷ Bind *var* to the lexical compilation environment.

**&aux** *var*<sup>\*</sup>

▷ Bind *vars* as in **let\***.

## 9.5 Control Flow

(*s*if *test* then [*else*<sub>NIL</sub>])

▷ Return values of *then* if *test* returns T; return values of *else* otherwise.

(*m*cond (*test* then<sup>P<sub>k</sub></sup> [*test*])<sup>\*</sup>)

▷ Return the values of the first *then*<sup>\*</sup> whose *test* returns T; return NIL if all *tests* return NIL.

$\left\{ \begin{array}{l} \text{mwhen} \\ \text{munless} \end{array} \right\} \text{test } \text{foo}^{\text{P}_k}$

▷ Evaluate *foos* and return their values if *test* returns T or NIL, respectively. Return NIL otherwise.

(*f*slot-boundp *instance slot*)    ▷ T if *slot* in *instance* is bound.

(*m*defclass *foo* (*superclass* <sup>standard-object</sup>)

$$\left( \begin{array}{l} \text{slot} \\ \left( \begin{array}{l} \text{:reader } \textit{reader}^* \\ \text{:writer } \left\{ \begin{array}{l} \textit{writer} \\ \text{(setf } \textit{writer}) \end{array} \right\}^* \\ \text{:accessor } \textit{accessor}^* \\ \text{:allocation } \left\{ \begin{array}{l} \text{:instance} \\ \text{:class } \textit{instance} \end{array} \right\}^* \\ \text{:initarg } [:\textit{initarg-name}]^* \\ \text{:initform } \textit{form} \\ \text{:type } \textit{type} \\ \text{:documentation } \textit{slot-doc} \end{array} \right) \end{array} \right)$$

(:default-initargs {*name value*}\*)  
(:documentation *class-doc*)  
(:metaclass *name* <sup>standard-class</sup>)

▷ Define or modify class *foo* as a subclass of *superclasses*. Transform existing instances, if any, by *g*make-instances-obsolete. In a new instance *i* of *foo*, a *slot*'s value defaults to *form* unless set via *[:initarg-name]*; it is readable via (*reader i*) or (*accessor i*), and writable via (*writer value i*) or (**setf** (*accessor i*) *value*). *slots* with **:allocation :class** are shared by all instances of class *foo*.

(*f*find-class *symbol* [*errorp*] [*environment*])  
▷ Return class named *symbol*. **setfable**.

(*g*make-instance *class* {*[:initarg value]*\* *other-keyarg*\*)  
▷ Make new instance of *class*.

(*g*reinitialize-instance *instance* {*[:initarg value]*\* *other-keyarg*\*)  
▷ Change local slots of *instance* according to *initargs* by means of *g*shared-initialize.

(*f*slot-value *foo slot*)    ▷ Return value of *slot* in *foo*. **setfable**.

(*f*slot-makunbound *instance slot*)  
▷ Make *slot* in *instance* unbound.

$\left( \begin{array}{l} \text{mwith-slots } (\widehat{\text{slot}} (\widehat{\text{var}} \textit{slot})^*) \\ \text{mwith-accessors } (\widehat{\text{var}} \textit{accessor})^* \end{array} \right) \textit{instance} (\text{declare } \widehat{\text{decl}})^* \textit{form}^*$   
▷ Return values of *forms* after evaluating them in a lexical environment with slots of *instance* visible as **setfable slots** or *vars*/with *accessors* of *instance* visible as **setfable vars**.

(*g*class-name *class*)  
(**setf** *g*class-name) *new-name class*)    ▷ Get/set name of *class*.

(*f*class-of *foo*)    ▷ Class *foo* is a direct instance of.

(*g*change-class *instance new-class* {*[:initarg value]*\* *other-keyarg*\*)  
▷ Change class of *instance* to *new-class*. Retain the status of any slots that are common between *instance*'s original class and *new-class*. Initialize any newly added slots with the values of the corresponding *initargs* if any, or with the values of their **:initform** forms if not.

(*g*make-instances-obsolete *class*)  
▷ Update all existing instances of *class* using *g*update-instance-for-redefined-class.

$\left( \begin{array}{l} \text{ginitialize-instance } \textit{instance} \\ \text{gupdate-instance-for-different-class } \textit{previous current} \end{array} \right) \left\{ \begin{array}{l} \text{[:initarg value]}^* \textit{other-keyarg}^* \end{array} \right\}$   
▷ Set slots on behalf of *g*make-instance/of *g*change-class by means of *g*shared-initialize.

(*g*update-instance-for-redefined-class *new-instance added-slots discarded-slots discarded-slots-property-list* {*[:initarg value]*\* *other-keyarg*\*)  
▷ On behalf of *g*make-instances-obsolete and by means of *g*shared-initialize, set any *initarg* slots to their corresponding values; set any remaining *added-slots* to the values of their **:initform** forms. Not to be called by user.

## 9.6 Iteration

$\left( \begin{array}{l} \text{mdo} \\ \text{mdo*} \end{array} \right) \left( \begin{array}{l} \text{var} \\ (\text{var } [\textit{start} \textit{step}]) \end{array} \right)^* (\textit{stop result}^*) (\text{declare } \widehat{\text{decl}})^* \left( \begin{array}{l} \widehat{\text{tag}} \\ \text{form} \end{array} \right)^*$

▷ Evaluate *stagbody*-like body with *vars* successively bound according to the values of the corresponding *start* and *step* forms. *vars* are bound in parallel/sequentially, respectively. Stop iteration when *stop* is T. Return values of result\*. Implicitly, the whole form is a *stagbody* named NIL.

(*m*dotimes (*var i* [*result* nil]) (declare *decl*)\* {*tag*}*form*\*)  
▷ Evaluate *stagbody*-like body with *var* successively bound to integers from 0 to *i* - 1. Upon evaluation of *result*, *var* is *i*. Implicitly, the whole form is a *stagbody* named NIL.

(*m*dolist (*var list* [*result* nil]) (declare *decl*)\* {*tag*}*form*\*)  
▷ Evaluate *stagbody*-like body with *var* successively bound to the elements of *list*. Upon evaluation of *result*, *var* is NIL. Implicitly, the whole form is a *stagbody* named NIL.

## 9.7 Loop Facility

(*m*loop *form*\*)  
▷ **Simple Loop**. If *forms* do not contain any atomic Loop Facility keywords, evaluate them forever in an implicit *stagbody* named NIL.

(*m*loop *clause*\*)  
▷ **Loop Facility**. For Loop Facility keywords see below and Figure 1.

named *n* nil    ▷ Give *m*loop's implicit *stagbody* a name.

{with  $\left( \begin{array}{l} \text{var-s} \\ (\text{var-s}^*) \end{array} \right) [d\text{-type}] [= \textit{foo}]^{\dagger}$

{and  $\left( \begin{array}{l} \text{var-p} \\ (\text{var-p}^*) \end{array} \right) [d\text{-type}] [= \textit{bar}]^*$

where destructuring type specifier *d-type* has the form

{fixnum|float|T|NIL|{of-type  $\left( \begin{array}{l} \text{type} \\ (\text{type}^*) \end{array} \right) \}}$

▷ Initialize (possibly trees of) local variables *var-s* sequentially and *var-p* in parallel.

{for|as  $\left( \begin{array}{l} \text{var-s} \\ (\text{var-s}^*) \end{array} \right) [d\text{-type}]^{\dagger}$  {and  $\left( \begin{array}{l} \text{var-p} \\ (\text{var-p}^*) \end{array} \right) [d\text{-type}]^*$

▷ Begin of iteration control clauses. Initialize and step (possibly trees of) local variables *var-s* sequentially and *var-p* in parallel. Destructuring type specifier *d-type* as with **with**.

{upfrom|from|downfrom} *start*  
▷ Start stepping with *start*

{upto|downto|to|below|above} *form*  
▷ Specify *form* as the end value for stepping.

{in|on} *list*  
▷ Bind *var* to successive elements/tails, respectively, of *list*.

by {step|function} cdr  
▷ Specify the (positive) decrement or increment or the *function* of one argument returning the next part of the list.

= *foo* {then *bar* foo}  
▷ Bind *var* initially to *foo* and later to *bar*.

across *vector*  
▷ Bind *var* to successive elements of *vector*.

being {the|each}  
▷ Iterate over a hash table or a package.

{hash-key|hash-keys} {of|in} *hash-table* {using (hash-value *value*)}  
▷ Bind *var* successively to the keys of *hash-table*; bind *value* to corresponding values.





(*f*make-condition *condition-type* {[:*initarg-name value*]}\*)

▷ Return new instance of *condition-type*.

$\left\{ \begin{array}{l} \text{fsignal} \\ \text{fwarn} \\ \text{ferror} \end{array} \right\} \left\{ \begin{array}{l} \text{condition} \\ \text{condition-type } \{[:\text{initarg-name value}]\}^* \\ \text{control arg}^* \end{array} \right\}$

▷ Unless handled, signal as **condition**, **warning** or **error**, respectively, *condition* or a new instance of *condition-type* or, with *f*format *control* and *args* (see page 36), **simple-condition**, **simple-warning**, or **simple-error**, respectively. From *f*signal and *f*warn, return NIL.

(*f*cerror *continue-control*

$\left\{ \begin{array}{l} \text{condition } \text{continue-arg}^* \\ \text{condition-type } \{[:\text{initarg-name value}]\}^* \\ \text{control arg}^* \end{array} \right\}$

▷ Unless handled, signal as correctable **error** *condition* or a new instance of *condition-type* or, with *f*format *control* and *args* (see page 36), **simple-error**. In the debugger, use *f*format arguments *continue-control* and *continue-args* to tag the continue option. Return NIL.

(*m*ignore-errors *form*<sup>P</sup>)

▷ Return values of forms or, in case of **errors**, NIL and the condition.

(*f*invoke-debugger *condition*)

▷ Invoke debugger with *condition*.

(*m*assert *test* [(*place*\*)

$\left\{ \begin{array}{l} \text{condition } \text{continue-arg}^* \\ \text{condition-type } \{[:\text{initarg-name value}]\}^* \\ \text{control arg}^* \end{array} \right\}$ )]

▷ If *test*, which may depend on *places*, returns NIL, signal as correctable **error** *condition* or a new instance of *condition-type* or, with *f*format *control* and *args* (see page 36), **error**. When using the debugger's continue option, *places* can be altered before re-evaluation of *test*. Return NIL.

(*m*handler-case *foo* (*type* ([*var*]) (declare  $\widehat{\text{decl}}^*$ )<sup>P</sup> *condition-form*<sup>P</sup>)\*

[(*no-error* (*ord-λ*\*) (declare  $\widehat{\text{decl}}^*$ )<sup>P</sup> *form*<sup>P</sup>)]

▷ If, on evaluation of *foo*, a condition of *type* is signalled, evaluate matching *condition-forms* with *var* bound to the condition, and return their values. Without a condition, bind *ord-λs* to values of *foo* and return values of *forms* or, without a **no-error** clause, return values of *foo*. See page 17 for (*ord-λ*\*)<sup>P</sup>.

(*m*handler-bind ((*condition-type handler-function*)\*) *form*<sup>P</sup>)

▷ Return values of forms after evaluating them with *condition-types* dynamically bound to their respective *handler-functions* of argument condition.

(*m*with-simple-restart ( $\left\{ \begin{array}{l} \text{restart} \\ \text{NIL} \end{array} \right\}$  *control arg*\*) *form*<sup>P</sup>)

▷ Return values of forms unless *restart* is called during their evaluation. In this case, describe *restart* using *f*format *control* and *args* (see page 36) and return NIL and T.

(*m*restart-case *form* (*restart* (*ord-λ*\*)  $\left\{ \begin{array}{l} \text{:interactive } \text{arg-function} \\ \text{:report } \left\{ \begin{array}{l} \text{report-function} \\ \text{string } \underline{\text{restart}} \end{array} \right\} \\ \text{:test } \text{test-function} \end{array} \right\}$ )

(declare  $\widehat{\text{decl}}^*$ )<sup>P</sup> *restart-form*<sup>P</sup>)\*

▷ Return values of form or, if during evaluation of *form* one of the dynamically established *restarts* is called, the values of its restart-forms. A *restart* is visible under *condition* if (*funcall* #'*test-function condition*) returns T. If presented in the debugger, *restarts* are described by *string* or by #'*report-function* (of a stream). A *restart* can be called by (*invoke-restart restart arg*\*)<sup>P</sup>, where *args* match *ord-λ*\*, or by (*invoke-restart-interactively restart*) where a list of the respective *args* is supplied by #'*arg-function*. See page 17 for *ord-λ*.\*

(*g*allocate-instance *class* {[:*initarg value*]}\* *other-keyarg*\*)

▷ Return uninitialized instance of *class*. Called by *g*make-instance.

(*g*shared-initialize *instance*  $\left\{ \begin{array}{l} \text{initform-slots} \\ \text{T} \end{array} \right\}$  {[:*initarg-slot value*]}\*

*other-keyarg*\*)

▷ Fill the *initarg-slots* of *instance* with the corresponding *values*, and fill those *initform-slots* that are not *initarg-slots* with the values of their *initform* forms.

(*g*slot-missing *class instance slot*  $\left\{ \begin{array}{l} \text{setf} \\ \text{slot-boundp} \\ \text{slot-makunbound} \\ \text{slot-value} \end{array} \right\}$  [*value*])

(*g*slot-unbound *class instance slot*)

▷ Called on attempted access to non-existing or unbound *slot*. Default methods signal **error/unbound-slot**, respectively. Not to be called by user.

## 10.2 Generic Functions

(*f*next-method-p) ▷ T if enclosing method has a next method.

(*m*defgeneric  $\left\{ \begin{array}{l} \text{foo} \\ \text{(setf foo)} \end{array} \right\}$  (*required-var*\* [*&optional*  $\left\{ \begin{array}{l} \text{var} \\ \text{(var)} \end{array} \right\}^*$ ]

[*&rest var*] [*&key*  $\left\{ \begin{array}{l} \text{var} \\ \text{(var) (:key var)} \end{array} \right\}^*$ ] [*&allow-other-keys*])

$\left\{ \begin{array}{l} \text{:argument-precedence-order } \text{required-var}^+ \\ \text{(declare (optimize method-selection-optimization})^+)} \\ \text{:documentation } \text{string} \\ \text{:generic-function-class } \text{gf-class} \underline{\text{standard-generic-function}} \\ \text{:method-class } \text{method-class} \underline{\text{standard-method}} \\ \text{:method-combination } \text{c-type} \underline{\text{standard}} \text{ c-arg}^* \\ \text{(method } \text{defmethod-args})^* \end{array} \right\}$

▷ Define or modify generic function *foo*. Remove any methods previously defined by *defgeneric*. *gf-class* and the lambda parameters *required-var*\* and *var*\* must be compatible with existing methods. *defmethod-args* resemble those of *m*defmethod. For *c-type* see section 10.3.

(*f*ensure-generic-function  $\left\{ \begin{array}{l} \text{foo} \\ \text{(setf foo)} \end{array} \right\}$

$\left\{ \begin{array}{l} \text{:argument-precedence-order } \text{required-var}^+ \\ \text{:declare (optimize method-selection-optimization)} \\ \text{:documentation } \text{string} \\ \text{:generic-function-class } \text{gf-class} \\ \text{:method-class } \text{method-class} \\ \text{:method-combination } \text{c-type } \text{c-arg}^* \\ \text{:lambda-list } \text{lambda-list} \\ \text{:environment } \text{environment} \end{array} \right\}$

▷ Define or modify generic function *foo*. *gf-class* and *lambda-list* must be compatible with a pre-existing generic function or with existing methods, respectively. Changes to *method-class* do not propagate to existing methods. For *c-type* see section 10.3.

(*m*defmethod  $\left\{ \begin{array}{l} \text{foo} \\ \text{(setf foo)} \end{array} \right\}$   $\left\{ \begin{array}{l} \text{:before} \\ \text{:after} \\ \text{:around} \end{array} \right\}$  [*primary method*]

$\left\{ \begin{array}{l} \text{var} \\ \text{(spec-var } \left\{ \begin{array}{l} \text{class} \\ \text{(eql bar)} \end{array} \right\})^* \end{array} \right\}$  [*&optional*

$\left\{ \begin{array}{l} \text{var} \\ \text{(var [init [supplied-p]])} \end{array} \right\}$ ] [*&rest var*] [*&key*

$\left\{ \begin{array}{l} \text{var} \\ \text{(var } \left\{ \begin{array}{l} \text{var} \\ \text{(key var)} \end{array} \right\})^* \end{array} \right\}$ ] [*init [supplied-p]*]] [*&allow-other-keys*]

[*&aux*  $\left\{ \begin{array}{l} \text{var} \\ \text{(var [init])} \end{array} \right\}^*$ ]  $\left\{ \begin{array}{l} \text{(declare } \widehat{\text{decl}}^* \text{)}^* \\ \widehat{\text{doc}} \end{array} \right\}$  *form*<sup>P</sup>)



▷ Define **new method** for generic function *foo*. *spec-vars* specialize to either being of *class* or being **eq** *bar*, respectively. On invocation, *vars* and *spec-vars* of the **new method** act like parameters of a function with body *form\**. *forms* are enclosed in an implicit **block** *foo*. Applicable *qualifiers* depend on the **method-combination** type; see section 10.3.

$\left\{ \begin{array}{l} \text{gadd-method} \\ \text{gremove-method} \end{array} \right\} \text{generic-function method}$

▷ Add (if necessary) or remove (if any) *method* to/from *generic-function*.

$\text{gfind-method } \text{generic-function } \text{qualifiers } \text{specializers } [\text{error}\underline{\text{T}}]$

▷ Return suitable *method*, or signal **error**.

$\text{gcompute-applicable-methods } \text{generic-function } \text{args}$

▷ List of methods suitable for *args*, most specific first.

$\text{fcall-next-method } \text{arg}^* \text{current-args}$

▷ From within a method, call next method with *args*; return its values.

$\text{gno-applicable-method } \text{generic-function } \text{arg}^*$

▷ Called on invocation of *generic-function* on *args* if there is no applicable method. Default method signals **error**. Not to be called by user.

$\left\{ \begin{array}{l} \text{finvalid-method-error } \text{method} \\ \text{fmethod-combination-error} \end{array} \right\} \text{control } \text{arg}^*$

▷ Signal **error** on applicable method with invalid qualifiers, or on method combination. For *control* and *args* see **format**, page 36.

$\text{gno-next-method } \text{generic-function } \text{method } \text{arg}^*$

▷ Called on invocation of **call-next-method** when there is no next method. Default method signals **error**. Not to be called by user.

$\text{gfunction-keywords } \text{method}$

▷ Return list of keyword parameters of *method* and **T** if other keys are allowed.

$\text{gmethod-qualifiers } \text{method}$  ▷ List of qualifiers of *method*.

## 10.3 Method Combination Types

### standard

▷ Evaluate most specific **:around** method supplying the values of the generic function. From within this method, **fcall-next-method** can call less specific **:around** methods if there are any. If not, or if there are no **:around** methods at all, call all **:before** methods, most specific first, and the most specific primary method which supplies the values of the calling **fcall-next-method** if any, or of the generic function; and which can call less specific primary methods via **fcall-next-method**. After its return, call all **:after** methods, least specific first.

**and|or|append|list|nconc|progn|max|min|+**

▷ Simple built-in **method-combination** types; have the same usage as the *c-types* defined by the short form of **mdefine-method-combination**.

$\text{(mdefine-method-combination } \text{c-type}$

$\left\{ \begin{array}{l} \text{:documentation } \text{string} \\ \text{:identity-with-one-argument } \text{bool}\underline{\text{T}} \\ \text{:operator } \text{operator}\underline{\text{c-type}} \end{array} \right\}$

▷ **Short Form.** Define new **method-combination** *c-type*. In a generic function using *c-type*, evaluate most specific **:around** method supplying the values of the generic function. From within this method, **fcall-next-method** can call less specific **:around** methods if there are any. If not, or if there are no **:around** methods at all, return from the calling **call-next-method** or from the generic function, respectively, the values of (*operator* (*primary-method* *gen-arg\**)\*), *gen-arg\** being the arguments of the generic function. The *primary-methods* are ordered  $\left[ \begin{array}{l} \text{:most-specific-first} \\ \text{:most-specific-last} \end{array} \right] \underline{\text{[most-specific-first]}}$  (specified as *c-arg* in **mdefgeneric**). Using *c-type* as the *qualifier* in **mdefmethod** makes the method primary.

$\text{(mdefine-method-combination } \text{c-type } (\text{ord-}\lambda^*) ((\text{group}$

$\left. \begin{array}{l} \text{*} \\ \text{(qualifier* } \underline{\text{[*]}}) \\ \text{predicate} \end{array} \right\}$   
 $\left. \begin{array}{l} \text{:description } \text{control} \\ \text{:order } \left\{ \begin{array}{l} \text{:most-specific-first} \\ \text{:most-specific-last} \end{array} \right\} \underline{\text{[most-specific-first]}} \\ \text{:required } \text{bool} \end{array} \right\} \text{)*}$   
 $\left. \begin{array}{l} \text{(:arguments } \text{method-combination-}\lambda^*) \\ \text{(:generic-function } \text{symbol}) \\ \left\{ \begin{array}{l} \text{(declare } \underline{\text{decl*}}) \\ \text{doc} \end{array} \right\} \end{array} \right\} \text{body}^{\text{R}}$

▷ **Long Form.** Define new **method-combination** *c-type*. A call to a generic function using *c-type* will be equivalent to a call to the forms returned by *body\** with *ord-λ\** bound to *c-arg\** (cf. **mdefgeneric**), with *symbol* bound to the generic function, with *method-combination-λ\** bound to the arguments of the generic function, and with *groups* bound to lists of methods. An applicable method becomes a member of the left-most *group* whose *predicate* or *qualifiers* match. Methods can be called via **mcall-method**. Lambda lists (*ord-λ\**) and (*method-combination-λ\**) according to *ord-λ* on page 17, the latter enhanced by an optional **&whole** argument.

$\text{(mcall-method}$

$\left\{ \begin{array}{l} \text{method} \\ \text{(mmake-method } \underline{\text{form}}) \end{array} \right\} \left[ \left( \left\{ \text{next-method } \underline{\text{form}} \right\} \right)^* \right]$

▷ From within an effective method form, call *method* with the arguments of the generic function and with information about its *next-methods*; return its values.

## 11 Conditions and Errors

For standardized condition types cf. Figure 2 on page 31.

$\text{(mdefine-condition } \text{foo } (\text{parent-type}^* \underline{\text{condition}})$

$\left. \begin{array}{l} \text{slot} \\ \left\{ \begin{array}{l} \text{:reader } \text{reader}^* \\ \text{:writer } \left\{ \begin{array}{l} \text{writer} \\ \text{(setf } \text{writer}) \end{array} \right\}^* \\ \text{:accessor } \text{accessor}^* \\ \text{:allocation } \left\{ \begin{array}{l} \text{:instance} \\ \text{:class} \end{array} \right\} \underline{\text{[instance]}} \\ \text{:initarg } \left[ \text{:initarg-name} \right]^* \\ \text{:initform } \text{form} \\ \text{:type } \text{type} \\ \text{:documentation } \text{slot-doc} \end{array} \right\} \end{array} \right\} \text{)*}$   
 $\left\{ \begin{array}{l} \text{:default-initargs } \{ \text{name value} \}^* \\ \text{:documentation } \text{condition-doc} \end{array} \right\}$   
 $\left\{ \begin{array}{l} \text{:report } \left\{ \begin{array}{l} \text{string} \\ \text{report-function} \end{array} \right\} \end{array} \right\}$

▷ Define, as a subtype of *parent-types*, condition type *foo*. In a new condition, a *slot*'s value defaults to *form* unless set via  $\text{[:initarg } \text{name}]$ ; it is readable via (*reader* *i*) or (*accessor* *i*), and writable via (*writer* *value* *i*) or (**setf** (*accessor* *i*) *value*). With **:allocation** **:class**, *slot* is shared by all conditions of type *foo*. A condition is reported by *string* or by *report-function* of arguments condition and stream.

## 13 Input/Output

### 13.1 Predicates

(*f*stream-p *foo*)

(*f*pathname-p *foo*) ▷ T if *foo* is of indicated type.

(*f*readtable-p *foo*)

(*f*input-stream-p *stream*)

(*f*output-stream-p *stream*)

(*f*interactive-stream-p *stream*)

(*f*open-stream-p *stream*)

▷ Return T if *stream* is for input, for output, interactive, or open, respectively.

(*f*pathname-match-p *path wildcard*)

▷ T if *path* matches *wildcard*.

(*f*wild-pathname-p *path* [{:host|:device|:directory|:name|:type|:version|NIL}])

▷ Return T if indicated component in *path* is wildcard. (NIL indicates any component.)

### 13.2 Reader

{*f*y-or-n-p  
*f*yes-or-no-p} [*control arg\**])

▷ Ask user a question and return T or NIL depending on their answer. See page 36, *f*format, for *control* and *args*.

(*m*with-standard-io-syntax *form<sup>P</sup>\**)

▷ Evaluate *forms* with standard behaviour of reader and printer. Return values of forms.

{*f*read  
*f*read-preserving-whitespace} [*stream* *v*\*standard-input\*] [*eof-err* *eof-val* *recursive*]]]]

▷ Read printed representation of object.

(*f*read-from-string *string* [*eof-error* *eof-val*]]

{[:start *start*  
:end *end*  
:preserve-whitespace *bool*]]]]

▷ Return object read from string and zero-indexed position of next character.

(*f*read-delimited-list *char* [*stream* *v*\*standard-input\*] [*recursive*]]])

▷ Continue reading until encountering *char*. Return list of objects read. Signal error if no *char* is found in stream.

(*f*read-char [*stream* *v*\*standard-input\*] [*eof-err* *eof-val* *recursive*]]])

▷ Return next character from *stream*.

(*f*read-char-no-hang [*stream* *v*\*standard-input\*] [*eof-error* *eof-val* *recursive*]]])

▷ Next character from *stream* or NIL if none is available.

(*f*peek-char [*mode* *stream* *v*\*standard-input\*] [*eof-error* *eof-val* *recursive*]]])

▷ Next, or if *mode* is T, next non-whitespace character, or if *mode* is a character, next instance of it, from *stream* without removing it there.

(*f*unread-char *character* [*stream* *v*\*standard-input\*])

▷ Put last *f*read-chared *character* back into *stream*; return NIL.

(*f*read-byte *stream* [*eof-err* *eof-val*]]])

▷ Read next byte from binary *stream*.

(*f*read-line [*stream* *v*\*standard-input\*] [*eof-err* *eof-val* *recursive*]]])

▷ Return a line of text from *stream* and T if line has been ended by end of file.

(*m*restart-bind (( $\widehat{\text{restart}}$   
NIL}) *restart-function*

{[:interactive-function *arg-function*  
:report-function *report-function*  
:test-function *test-function*})\*) *form<sup>P</sup>\**)

▷ Return values of forms evaluated with dynamically established *restarts* whose *restart-functions* should perform a non-local transfer of control. A restart is visible under *condition* if (*test-function condition*) returns T. If presented in the debugger, *restarts* are described by *restart-function* (of a stream). A *restart* can be called by (*invoke-restart restart arg\**), where *args* must be suitable for the corresponding *restart-function*, or by (*invoke-restart-interactively restart*) where a list of the respective *args* is supplied by *arg-function*.

(*f*invoke-restart *restart arg\**)

(*f*invoke-restart-interactively *restart*)

▷ Call function associated with *restart* with arguments given or prompted for, respectively. If *restart* function returns, return its values.

{*f*find-restart  
*f*compute-restarts *name*} [*condition*])

▷ Return innermost restart name, or a list of all restarts, respectively, out of those either associated with *condition* or un-associated at all; or, without *condition*, out of all restarts. Return NIL if search is unsuccessful.

(*f*restart-name *restart*) ▷ Name of restart.

{*f*abort  
*f*muffle-warning  
*f*continue  
*f*store-value *value*  
*f*use-value *value*} [*condition* *recursive*]]]

▷ Transfer control to innermost applicable restart with same name (i.e. **abort**, ..., **continue** ...) out of those either associated with *condition* or un-associated at all; or, without *condition*, out of all restarts. If no restart is found, signal **control-error** for *f*abort and *f*muffle-warning, or return NIL for the rest.

(*m*with-condition-restarts *condition restarts form<sup>P</sup>\**)

▷ Evaluate *forms* with *restarts* dynamically associated with *condition*. Return values of forms.

(*f*arithmetic-error-operation *condition*)

(*f*arithmetic-error-operands *condition*)

▷ List of function or of its operands respectively, used in the operation which caused *condition*.

(*f*cell-error-name *condition*)

▷ Name of cell which caused *condition*.

(*f*unbound-slot-instance *condition*)

▷ Instance with unbound slot which caused *condition*.

(*f*print-not-readable-object *condition*)

▷ The object not readably printable under *condition*.

(*f*package-error-package *condition*)

(*f*file-error-pathname *condition*)

(*f*stream-error-stream *condition*)

▷ Package, path, or stream, respectively, which caused the *condition* of indicated type.

(*f*type-error-datum *condition*)

(*f*type-error-expected-type *condition*)

▷ Object which caused *condition* of type **type-error**, or its expected type, respectively.

(*f*simple-condition-format-control *condition*)

(*f*simple-condition-format-arguments *condition*)

▷ Return *f*format control or list of *f*format arguments, respectively, of *condition*.

*v*\*break-on-signals\* *recursive*]]]

▷ Condition type debugger is to be invoked on.



- (*f*pprint-newline  $\left. \begin{array}{l} \text{:linear} \\ \text{:fill} \\ \text{:miser} \\ \text{:mandatory} \end{array} \right\} [stream \underline{v*standard-output*}])$
- ▷ Print a conditional newline if *stream* is a pretty printing stream. Return NIL.
- v*\*print-array\*   ▷ If T, print arrays *f*readably.
- v*\*print-base\*<sub>[T]</sub>   ▷ Radix for printing rationals, from 2 to 36.
- v*\*print-case\*<sub>[upcase]</sub>
- ▷ Print symbol names all uppercase (:upcase), all lowercase (:downcase), capitalized (:capitalize).
- v*\*print-circle\*<sub>[NIL]</sub>
- ▷ If T, avoid indefinite recursion while printing circular structure.
- v*\*print-escape\*<sub>[NIL]</sub>
- ▷ If NIL, do not print escape characters and package prefixes.
- v*\*print-gensym\*<sub>[NIL]</sub>   ▷ If T, print #: before uninterned symbols.
- v*\*print-length\*<sub>[NIL]</sub>
- v*\*print-level\*<sub>[NIL]</sub>
- v*\*print-lines\*<sub>[NIL]</sub>
- ▷ If integer, restrict printing of objects to that number of elements per level/to that depth/to that number of lines.
- v*\*print-miser-width\*
- ▷ If integer and greater than the width available for printing a substructure, switch to the more compact miser style.
- v*\*print-pretty\*   ▷ If T, print prettily.
- v*\*print-radix\*<sub>[NIL]</sub>   ▷ If T, print rationals with a radix indicator.
- v*\*print-readably\*<sub>[NIL]</sub>
- ▷ If T, print *f*readably or signal error **print-not-readable**.
- v*\*print-right-margin\*<sub>[NIL]</sub>
- ▷ Right margin width in ems while pretty-printing.
- (*f*set-pprint-dispatch *type function* [*priority*<sub>[T]</sub> [*table*<sub>[v\*print-pprint-dispatch\*]</sub>]])
- ▷ Install entry comprising *function* of arguments stream and object to print; and *priority* as *type* into *table*. If *function* is NIL, remove *type* from *table*. Return NIL.
- (*f*pprint-dispatch *foo* [*table*<sub>[v\*print-pprint-dispatch\*]</sub>])
- ▷ Return highest priority *function* associated with type of *foo* and T if there was a matching type specifier in *table*.
- (*f*copy-pprint-dispatch [*table*<sub>[v\*print-pprint-dispatch\*]</sub>])
- ▷ Return copy of *table* or, if *table* is NIL, initial value of *v\*print-pprint-dispatch\**.
- v*\*print-pprint-dispatch\*   ▷ Current pretty print dispatch table.

### 13.5 Format

- (*m*formatter *control*)
- ▷ Return function of *stream* and *arg\** applying *f*format to *stream*, *control*, and *arg\** returning NIL or any excess *args*.
- (*f*format {T|NIL|out-string|out-stream} *control arg\**)
- ▷ Output string *control* which may contain ~ directives possibly taking some *args*. Alternatively, *control* can be a function returned by *m*formatter which is then applied to *out-stream* and *arg\**. Output to *out-string*, *out-stream* or, if first argument is T, to *v\*standard-output\**. Return NIL. If first argument is NIL, return formatted output.

- (*f*read-sequence *sequence stream* [:start *start*<sub>[T]</sub>][:end *end*<sub>[NIL]</sub>])
- ▷ Replace elements of *sequence* between *start* and *end* with elements from binary or character *stream*. Return index of *sequence*'s first unmodified element.
- (*f*readtable-case *readtable*)<sub>[upcase]</sub>
- ▷ Case sensitivity attribute (one of :upcase, :downcase, :preserve, :invert) of *readtable*. **setfable**.
- (*f*copy-readtable [*from-readtable*<sub>[v\*readtable\*]</sub> [*to-readtable*<sub>[NIL]</sub>]])
- ▷ Return copy of *from-readtable*.
- (*f*set-syntax-from-char *to-char from-char* [*to-readtable*<sub>[v\*readtable\*]</sub> [*from-readtable*<sub>[standard-readtable]</sub>]])
- ▷ Copy syntax of *from-char* to *to-readtable*. Return T.
- v*\*readtable\*   ▷ Current readtable.
- v*\*read-base\*<sub>[T]</sub>   ▷ Radix for reading integers and ratios.
- v*\*read-default-float-format\*<sub>[single-float]</sub>
- ▷ Floating point format to use when not indicated in the number read.
- v*\*read-suppress\*<sub>[NIL]</sub>
- ▷ If T, reader is syntactically more tolerant.
- (*f*set-macro-character *char function* [*non-term-p*<sub>[NIL]</sub> [*rt*<sub>[v\*readtable\*]</sub>]])
- ▷ Make *char* a macro character associated with *function* of stream and *char*. Return T.
- (*f*get-macro-character *char* [*rt*<sub>[v\*readtable\*]</sub>])
- ▷ Reader macro function associated with *char*, and T if *char* is a non-terminating macro character.
- (*f*make-dispatch-macro-character *char* [*non-term-p*<sub>[NIL]</sub> [*rt*<sub>[v\*readtable\*]</sub>]])
- ▷ Make *char* a dispatching macro character. Return T.
- (*f*set-dispatch-macro-character *char sub-char function* [*rt*<sub>[v\*readtable\*]</sub>])
- ▷ Make *function* of stream, *n*, *sub-char* a dispatch function of *char* followed by *n*, followed by *sub-char*. Return T.
- (*f*get-dispatch-macro-character *char sub-char* [*rt*<sub>[v\*readtable\*]</sub>])
- ▷ Dispatch function associated with *char* followed by *sub-char*.

### 13.3 Character Syntax

- #| *multi-line-comment\** |#  
; *one-line-comment\**
- ▷ Comments. There are stylistic conventions:
- |                       |  |
|-----------------------|--|
| ;;; <i>title</i>      | ▷ Short title for a block of code.       |
| ;; <i>intro</i>       | ▷ Description before a block of code.    |
| ;; <i>state</i>       | ▷ State of program or of following code. |
| ; <i>explanation</i>  | ▷ Regarding line on which it appears.    |
| ; <i>continuation</i> |  |
- (*foo\** [ *bar*<sub>[NIL]</sub> ])   ▷ List of *foos* with the terminating cdr *bar*.
- "   ▷ Begin and end of a string.
- '*foo*   ▷ (*quote foo*); *foo* unevaluated.
- `([*foo*] [*bar*] [*@baz*] [*..quux*] [*bing*])
- ▷ Backquote. *quote foo* and *bing*; evaluate *bar* and splice the lists *baz* and *quux* into their elements. When nested, outermost commas inside the innermost backquote expression belong to this backquote.
- #\c   ▷ (*f*character "c"), the character *c*.
- #B*n*; #O*n*; *n*.; #X*n*; #rR*n*
- ▷ Integer of radix 2, 8, 10, 16, or *r*;  $2 \leq r \leq 36$ .



$n/d$  ▷ The **ratio**  $\frac{n}{d}$ .

$\{[m].n[\{S|F|D|L|E\}x_{\text{E0}}]|m[.[n]]\{S|F|D|L|E\}x\}$   
 ▷  $m.n \cdot 10^x$  as **short-float**, **single-float**, **double-float**, **long-float**,  
 or the type from **\*read-default-float-format\***.

**#C**( $a$   $b$ ) ▷ ( $f$ **complex**  $a$   $b$ ), the complex number  $a + bi$ .

**#'** $foo$  ▷ ( $s$ **function**  $foo$ ); the function named  $foo$ .

**#n****A** $sequence$  ▷  $n$ -dimensional array.

**#**[ $n$ ]( $foo^*$ )  
 ▷ Vector of some (or  $n$ )  $foos$  filled with last  $foo$  if necessary.

**#**[ $n$ ]\* $b^*$   
 ▷ Bit vector of some (or  $n$ )  $bs$  filled with last  $b$  if necessary.

**#S**( $type$  { $slot$   $value$ } $^*$ ) ▷ Structure of  $type$ .

**#P** $string$  ▷ A pathname.

**#:** $foo$  ▷ Unintended symbol  $foo$ .

**#.** $form$  ▷ Read-time value of  $form$ .

**v\*****read-eval\*** $\square$  ▷ If NIL, a **reader-error** is signalled at **#.**

**#integer=**  $foo$  ▷ Give  $foo$  the label  $integer$ .

**#integer#** ▷ Object labelled  $integer$ .

**#<** ▷ Have the reader signal **reader-error**.

**#+feature**  $when$ - $feature$

**#-feature**  $unless$ - $feature$

▷ Means  $when$ - $feature$  if  $feature$  is T; means  $unless$ - $feature$  if  $feature$  is NIL.  $feature$  is a symbol from **v\*features\***, or (**{and** |**or**}  $feature^*$ ), or (**not**  $feature$ ).

**v\*features\***

▷ List of symbols denoting implementation-dependent features.

| $c^*$ |\  $c$

▷ Treat arbitrary character(s)  $c$  as alphabetic preserving case.

## 13.4 Printer

$\left( \begin{array}{l} f\text{prin1} \\ f\text{print} \\ f\text{pprint} \\ f\text{princ} \end{array} \right) foo$  [ $\widetilde{stream}$   $v*$ **standard-output\***]

▷ Print  $foo$  to  $stream$   $f$ **readably**,  $f$ **readably** between a newline and a space,  $f$ **readably** after a newline, or human-readably without any extra characters, respectively.  $f$ **prin1**,  $f$ **print** and  $f$ **princ** return  $foo$ .

( $f$ **prin1-to-string**  $foo$ )

( $f$ **princ-to-string**  $foo$ )

▷ Print  $foo$  to  $string$   $f$ **readably** or human-readably, respectively.

( $g$ **print-object**  $object$   $\widetilde{stream}$ )

▷ Print  $object$  to  $stream$ . Called by the Lisp printer.

( $m$ **print-unreadable-object** ( $foo$   $\widetilde{stream}$   $\left\{ \begin{array}{l} :type\ bool\ \square \\ :identity\ bool\ \square \end{array} \right\}$ )  $form^P_k$ )

▷ Enclosed in **#<** and **>**, print  $foo$  by means of  $forms$  to  $stream$ . Return **NIL**.

( $f$ **terpri** [ $\widetilde{stream}$   $v*$ **standard-output\***])

▷ Output a newline to  $stream$ . Return **NIL**.

( $f$ **fresh-line** [ $\widetilde{stream}$   $v*$ **standard-output\***])

▷ Output a newline to  $stream$  and return **T** unless  $stream$  is already at the start of a line.

( $f$ **write-char**  $char$  [ $\widetilde{stream}$   $v*$ **standard-output\***])  
 ▷ Output  $char$  to  $stream$ .

$\left( \begin{array}{l} f\text{write-string} \\ f\text{write-line} \end{array} \right) string$  [ $\widetilde{stream}$   $v*$ **standard-output\***] [ $\left\{ \begin{array}{l} :start\ start_{\square} \\ :end\ end_{\square} \end{array} \right\}$ ]

▷ Write  $string$  to  $stream$  without/with a trailing newline.

( $f$ **write-byte**  $byte$   $\widetilde{stream}$ ) ▷ Write  $byte$  to binary  $stream$ .

( $f$ **write-sequence**  $sequence$   $\widetilde{stream}$   $\left\{ \begin{array}{l} :start\ start_{\square} \\ :end\ end_{\square} \end{array} \right\}$ )

▷ Write elements of  $sequence$  to binary or character  $stream$ .

$\left( \begin{array}{l} f\text{write} \\ f\text{write-to-string} \end{array} \right) foo$   $\left\{ \begin{array}{l} :array\ bool \\ :base\ radix \\ :case\ \left\{ \begin{array}{l} :upcase \\ :downcase \\ :capitalize \end{array} \right\} \\ :circle\ bool \\ :escape\ bool \\ :gensym\ bool \\ :length\ \{int\ \text{NIL}\} \\ :level\ \{int\ \text{NIL}\} \\ :lines\ \{int\ \text{NIL}\} \\ :miser-width\ \{int\ \text{NIL}\} \\ :pprint-dispatch\ dispatch-table \\ :pretty\ bool \\ :radix\ bool \\ :readably\ bool \\ :right-margin\ \{int\ \text{NIL}\} \\ :stream\ \widetilde{stream}\ v*\text{standard-output*} \end{array} \right\}$

▷ Print  $foo$  to  $stream$  and return  $foo$ , or print  $foo$  into  $string$ , respectively, after dynamically setting printer variables corresponding to keyword parameters (**\*print-bar\*** becoming **:bar**). (**:stream** keyword with  $f$ **write** only.)

( $f$ **pprint-fill**  $\widetilde{stream}$   $foo$  [ $parenthesis_{\square}$ ] [ $noop$ ])

( $f$ **pprint-tabular**  $\widetilde{stream}$   $foo$  [ $parenthesis_{\square}$ ] [ $noop$ ] [ $n_{\square}$ ])

( $f$ **pprint-linear**  $\widetilde{stream}$   $foo$  [ $parenthesis_{\square}$ ] [ $noop$ ])

▷ Print  $foo$  to  $stream$ . If  $foo$  is a list, print as many elements per line as possible; do the same in a table with a column width of  $n$  ems; or print either all elements on one line or each on its own line, respectively. Return **NIL**. Usable with  $f$ **format** directive  $\sim//$ .

( $m$ **pprint-logical-block** ( $\widetilde{stream}$   $list$   $\left\{ \begin{array}{l} :prefix\ string \\ :per-line-prefix\ string \\ :suffix\ string_{\square} \end{array} \right\}$ )

(**declare**  $\widehat{decl}^*$ ) $^*$   $form^P_k$ )

▷ Evaluate  $forms$ , which should print  $list$ , with  $stream$  locally bound to a pretty printing stream which outputs to the original  $stream$ . If  $list$  is in fact not a list, it is printed by  $f$ **write**. Return **NIL**.

( $m$ **pprint-pop**)

▷ Take next element off  $list$ . If there is no remaining tail of  $list$ , or **v\*print-length\*** or **v\*print-circle\*** indicate printing should end, send element together with an appropriate indicator to  $stream$ .

( $f$ **pprint-tab**  $\left\{ \begin{array}{l} :line \\ :line-relative \\ :section \\ :section-relative \end{array} \right\} c$   $i$  [ $\widetilde{stream}$   $v*$ **standard-output\***])

▷ Move cursor forward to column number  $c + ki$ ,  $k \geq 0$  being as small as possible.

( $f$ **pprint-indent**  $\left\{ \begin{array}{l} :block \\ :current \end{array} \right\} n$  [ $\widetilde{stream}$   $v*$ **standard-output\***])

▷ Specify indentation for innermost logical block relative to leftmost position/to current position. Return **NIL**.

( $m$ **pprint-exit-if-list-exhausted**)

▷ If  $list$  is empty, terminate logical block. Return **NIL** otherwise.



(*f*close *stream* [:abort *bool*])  
 ▷ Close *stream*. Return *T* if *stream* had been open. If :abort is *T*, delete associated file.

(*m*with-open-file (*stream path open-arg\**) (declare *decl\**)*form\**)  
 ▷ Use *f*open with *open-args* to temporarily create *stream* to *path*; return values of forms.

(*m*with-open-stream (*foo stream*) (declare *decl\**)*form\**)  
 ▷ Evaluate *forms* with *foo* locally bound to *stream*. Return values of forms.

(*m*with-input-from-string (*foo string*  $\left\{ \begin{array}{l} \text{:index } \textit{index} \\ \text{:start } \textit{start} \\ \text{:end } \textit{end} \end{array} \right\}$ ) (declare *decl\**)*form\**)  
 ▷ Evaluate *forms* with *foo* locally bound to input **string-stream** from *string*. Return values of forms; store next reading position into *index*.

(*m*with-output-to-string (*foo*  $\left[ \begin{array}{l} \textit{string} \\ \textit{element-type } \textit{type} \end{array} \right]$ ) (declare *decl\**)*form\**)  
 ▷ Evaluate *forms* with *foo* locally bound to an output **string-stream**. Append output to *string* and return values of forms if *string* is given. Return string containing output otherwise.

(*f*stream-external-format *stream*)  
 ▷ External file format designator.

\*terminal-io\*    ▷ Bidirectional stream to user terminal.

\*standard-input\*

\*standard-output\*

\*error-output\*

▷ Standard input stream, standard output stream, or standard error output stream, respectively.

\*debug-io\*

\*query-io\*

▷ Bidirectional streams for debugging and user interaction.

## 13.7 Pathnames and Files

(*f*make-pathname

$$\left. \begin{array}{l} \text{:host } \{ \textit{host} | \text{NIL} | \text{:unspecific} \} \\ \text{:device } \{ \textit{device} | \text{NIL} | \text{:unspecific} \} \\ \text{:directory } \left\{ \begin{array}{l} \textit{directory} | \text{:wild} | \text{NIL} | \text{:unspecific} \\ \left\{ \begin{array}{l} \text{:absolute} \\ \text{:relative} \end{array} \right\} \left\{ \begin{array}{l} \textit{directory} \\ \text{:wild} \\ \text{:wild-inferiors} \\ \text{:up} \\ \text{:back} \end{array} \right\} \end{array} \right\} \\ \text{:name } \{ \textit{file-name} | \text{:wild} | \text{NIL} | \text{:unspecific} \} \\ \text{:type } \{ \textit{file-type} | \text{:wild} | \text{NIL} | \text{:unspecific} \} \\ \text{:version } \{ \text{:newest} | \textit{version} | \text{:wild} | \text{NIL} | \text{:unspecific} \} \\ \text{:defaults } \textit{path} \left[ \begin{array}{l} \textit{host} \textit{from} \textit{v} \textit{default-pathname-defaults} \\ \text{:local} | \text{:common} | \text{:local} \end{array} \right] \\ \text{:case } \{ \text{:local} | \text{:common} | \text{:local} \} \end{array} \right\}$$

▷ Construct a logical pathname if there is a logical pathname translation for *host*, otherwise construct a physical pathname. For :case :local, leave case of components unchanged. For :case :common, leave mixed-case components unchanged; convert all-uppercase components into local customary case; do the opposite with all-lowercase components.

$$\left\{ \begin{array}{l} \textit{pathname-host} \\ \textit{pathname-device} \\ \textit{pathname-directory} \\ \textit{pathname-name} \\ \textit{pathname-type} \end{array} \right\} \textit{path-or-stream} \left[ \text{:case } \left\{ \begin{array}{l} \text{:local} \\ \text{:common} | \text{:local} \end{array} \right\} \right]$$

(*f*pathname-version *path-or-stream*)

▷ Return pathname component.

~ [*min-col*] [*col-inc*] [*min-pad*] [*pad-char*]

[:] [*Q*] {*A*|*S*}

▷ **Aesthetic/Standard**. Print argument of any type for consumption by humans/by the reader, respectively. With :, print NIL as () rather than nil; with *Q*, add *pad-chars* on the left rather than on the right.

~ [*radix*] [*width*] [*pad-char*] [*comma-char*] [*comma-interval*]

[:] [*Q*] *R*

▷ **Radix**. (With one or more prefix arguments.) Print argument as number; with :, group digits *comma-interval* each; with *Q*, always prepend a sign.

{-*R*|~*R*|~*OR*|~*Q*:*R*}

▷ **Roman**. Take argument as number and print it as English cardinal number, as English ordinal number, as Roman numeral, or as old Roman numeral, respectively.

~ [*width*] [*pad-char*] [*comma-char*] [*comma-interval*]

[:] [*Q*] {*D*|*B*|*O*|*X*}

▷ **Decimal/Binary/Octal/Hexadecimal**. Print integer argument as number. With :, group digits *comma-interval* each; with *Q*, always prepend a sign.

~ [*width*] [*dec-digits*] [*shift*] [*overflow-char*] [*pad-char*]

[:] [*Q*] *F*

▷ **Fixed-Format Floating-Point**. With *Q*, always prepend a sign.

~ [*width*] [*dec-digits*] [*exp-digits*] [*scale-factor*] [*overflow-char*] [*pad-char*] [*exp-char*]

[:] [*Q*] {*E*|*G*}

▷ **Exponential/General Floating-Point**. Print argument as floating-point number with *dec-digits* after decimal point and *exp-digits* in the signed exponent. With ~*G*, choose either ~*E* or ~*F*. With *Q*, always prepend a sign.

~ [*dec-digits*] [*int-digits*] [*width*] [*pad-char*]

[:] [*Q*] *S*

▷ **Monetary Floating-Point**. Print argument as fixed-format floating-point number. With :, put sign before any padding; with *Q*, always prepend a sign.

{-*C*|~*C*|~*QC*|~*Q*:*C*}

▷ **Character**. Print, spell out, print in #\ syntax, or tell how to type, respectively, argument as (possibly non-printing) character.

{-(*text* ~)|~:(*text* ~)|~*Q*(*text* ~)|~*Q*:(*text* ~)}

▷ **Case-Conversion**. Convert *text* to lowercase, convert first letter of each word to uppercase, capitalize first word and convert the rest to lowercase, or convert to uppercase, respectively.

{-*P*|~*P*|~*QP*|~*Q*:*P*}

▷ **Plural**. If argument *eq* 1 print nothing, otherwise print *s*; do the same for the previous argument; if argument *eq* 1 print *y*, otherwise print *ies*; do the same for the previous argument, respectively.

~ [*n*] %    ▷ **Newline**. Print *n* newlines.

~ [*n*] &

▷ **Fresh-Line**. Print *n* – 1 newlines if output stream is at the beginning of a line, or *n* newlines otherwise.

{-*~*|~*~*|~*Q*|~*Q*:}

▷ **Conditional Newline**. Print a newline like **pprint-newline** with argument :linear, :fill, :miser, or :mandatory, respectively.

{-~|~|~*Q*|~*Q*:}

▷ **Ignored Newline**. Ignore newline, or whitespace following newline, or both, respectively.

~ [*n*] |    ▷ **Page**. Print *n* page separators.

~ [*n*] ~    ▷ **Tilde**. Print *n* tildes.

~ [*min-col*] [*col-inc*] [*min-pad*] [*pad-char*]

[:] [*Q*] < [*nl-text* ~[*spare*] [*width*]]:: {*text* ~;}<sup>\*</sup> *text* ~>

▷ **Justification**. Justify text produced by *texts* in a field of at least *min-col* columns. With :, right justify; with *Q*, left justify. If this would leave less than *spare* characters on the current line, output *nl-text* first.

~ [:] [C] < { [prefix<sub>opt</sub> ~:] | [per-line-prefix ~C:] } body [-; suffix<sub>opt</sub> ~:] [C] >

▷ **Logical Block.** Act like **pprint-logical-block** using *body* as *f***format** control string on the elements of the list argument or, with **C**, on the remaining arguments, which are extracted by **pprint-pop**. With **:**, *prefix* and *suffix* default to ( and ). When closed by ~C>, spaces in *body* are replaced with conditional newlines.

{~ [n<sub>opt</sub>] i | ~ [n<sub>opt</sub>] :i}

▷ **Indent.** Set indentation to *n* relative to leftmost/to current position.

~ [c<sub>opt</sub>] [,i<sub>opt</sub>] [:] [C] T

▷ **Tabulate.** Move cursor forward to column number *c* + *ki*, *k* ≥ 0 being as small as possible. With **:**, calculate column numbers relative to the immediately enclosing section. With **C**, move to column number *c*<sub>0</sub> + *c* + *ki* where *c*<sub>0</sub> is the current position.

{~ [m<sub>opt</sub>] \* | ~ [m<sub>opt</sub>] :\* | ~ [n<sub>opt</sub>] C\*}

▷ **Go-To.** Jump *m* arguments forward, or backward, or to argument *n*.

~ [limit] [:] [C] { text ~ }

▷ **Iteration.** Use *text* repeatedly, up to *limit*, as control string for the elements of the list argument or (with **C**) for the remaining arguments. With **:** or **C**:, list elements or remaining arguments should be lists of which a new one is used at each iteration step.

~ [x [,y [,z]]] ^

▷ **Escape Upward.** Leave immediately ~< ~>, ~< ~:>, ~{ ~}, ~?, or the entire *f***format** operation. With one to three prefixes, act only if *x* = 0, *x* = *y*, or *x* ≤ *y* ≤ *z*, respectively.

~ [i] [:] [C] [ [ {text ~;} \* text ] [~:; default] ~ ]

▷ **Conditional Expression.** Use the zero-indexed argument (or *i*th if given) *text* as a *f***format** control subclause. With **:**, use the first *text* if the argument value is NIL, or the second *text* if it is T. With **C**, do nothing for an argument value of NIL. Use the only *text* and leave the argument to be read again if it is T.

{~?|~C?}

▷ **Recursive Processing.** Process two arguments as control string and argument list, or take one argument as control string and use then the rest of the original arguments.

~ [prefix {,prefix}\*] [:] [C] / [package [:] [c1-user:] function /

▷ **Call Function.** Call all-uppercase *package::function* with the arguments *stream*, *format-argument*, *colon-p*, *at-sign-p* and *prefixes* for printing *format-argument*.

~ [:] [C] W

▷ **Write.** Print argument of any type obeying every printer control variable. With **:**, pretty-print. With **C**, print without limits on length or depth.

{V|#}

▷ In place of the comma-separated prefix parameters: use next argument or number of remaining unprocessed arguments, respectively.

## 13.6 Streams

(*f*open path {

- :direction { :input }
  - :output { :input }
  - :io { :input }
  - :probe { :input }
- :element-type { type }
  - :default { character }
- :if-exists { :new-version }
  - :error { :new-version if path specifies :newest; NIL otherwise }
  - :rename { :new-version if path specifies :newest; NIL otherwise }
  - :rename-and-delete { :new-version if path specifies :newest; NIL otherwise }
  - :overwrite { :new-version if path specifies :newest; NIL otherwise }
  - :append { :new-version if path specifies :newest; NIL otherwise }
  - :supersede { :new-version if path specifies :newest; NIL otherwise }
  - NIL { :new-version if path specifies :newest; NIL otherwise }
- :if-does-not-exist { :error }
  - :create { NIL for :direction :probe; {:create:error} otherwise }
  - NIL { NIL for :direction :probe; {:create:error} otherwise }
- :external-format format { default }

▷ Open file-stream to *path*.

(*f*make-concatenated-stream input-stream\*)

(*f*make-broadcast-stream output-stream\*)

(*f*make-two-way-stream input-stream-part output-stream-part)

(*f*make-echo-stream from-input-stream to-output-stream)

(*f*make-synonym-stream variable-bound-to-stream)

▷ Return stream of indicated type.

(*f*make-string-input-stream string [start<sub>opt</sub>] [end<sub>opt</sub>]])

▷ Return a string-stream supplying the characters from *string*.

(*f*make-string-output-stream [:element-type type character])

▷ Return a string-stream accepting characters (available via *f*get-output-stream-string).

(*f*concatenated-stream-streams concatenated-stream)

(*f*broadcast-stream-streams broadcast-stream)

▷ Return list of streams *concatenated-stream* still has to read from/*broadcast-stream* is broadcasting to.

(*f*two-way-stream-input-stream two-way-stream)

(*f*two-way-stream-output-stream two-way-stream)

(*f*echo-stream-input-stream echo-stream)

(*f*echo-stream-output-stream echo-stream)

▷ Return source stream or sink stream of *two-way-stream/echo-stream*, respectively.

(*f*synonym-stream-symbol synonym-stream)

▷ Return symbol of *synonym-stream*.

(*f*get-output-stream-string string-stream)

▷ Clear and return as a string characters on *string-stream*.

(*f*file-position stream { :start } { :end } position)

▷ Return position within stream, or set it to position and return **T** on success.

(*f*file-string-length stream foo)

▷ Length *foo* would have in *stream*.

(*f*listen [stream<sub>v,\*standard-input\*</sub>])

▷ **T** if there is a character in input *stream*.

(*f*clear-input [stream<sub>v,\*standard-input\*</sub>])

▷ Clear input from *stream*, return **NIL**.

{*f*clear-output } {*f*force-output } {*f*finish-output } [stream<sub>v,\*standard-output\*</sub>])

▷ End output to *stream* and return **NIL** immediately, after initiating flushing of buffers, or after flushing of buffers, respectively.

$\left\{ \begin{array}{l} \text{documentation} \\ (\text{setf } \underline{g} \text{ documentation}) \text{ new-doc} \end{array} \right\} \text{foo} \left\{ \begin{array}{l} \text{'variable'} | \text{'function'} \\ \text{'compiler-macro'} \\ \text{'method-combination'} \\ \text{'structure'} | \text{'type'} | \text{'setf'} | \text{T} \end{array} \right\}$

▷ Get/set documentation string of *foo* of given type.

**ct**

▷ Truth; the supertype of every type including **t**; the superclass of every class except **t**;  $\nu$ \***terminal-io\***.

**cnil**|c()

▷ Falsity; the empty list; the empty type, subtype of every type;  $\nu$ \***standard-input\***;  $\nu$ \***standard-output\***; the global environment.

## 14.4 Standard Packages

**common-lisp|cl**

▷ Exports the defined names of Common Lisp except for those in the **keyword** package.

**common-lisp-user|cl-user**

▷ Current package after startup; uses package **common-lisp**.

**keyword**

▷ Contains symbols which are defined to be of type **keyword**.

## 15 Compiler

### 15.1 Predicates

(*f* **special-operator-p** *foo*) ▷ **T** if *foo* is a special operator.

(*f* **compiled-function-p** *foo*)

▷ **T** if *foo* is of type **compiled-function**.

### 15.2 Compilation

(*f* **compile**  $\left\{ \begin{array}{l} \text{NIL definition} \\ \text{name} \\ (\text{setf name}) \end{array} \right\} [\text{definition}]$ )

▷ Return compiled function or replace *name*'s function definition with the compiled function. Return **T** in case of **warnings** or **errors**, and **T** in case of **warnings** or **errors** excluding **style-warnings**.

(*f* **compile-file** *file*  $\left\{ \begin{array}{l} \text{:output-file } \text{out-path} \\ \text{:verbose } \text{bool} [\nu \text{*compile-verbose*}] \\ \text{:print } \text{bool} [\nu \text{*compile-print*}] \\ \text{:external-format } \text{file-format} [\text{default}] \end{array} \right\}$ )

▷ Write compiled contents of *file* to *out-path*. Return **true** output path or **NIL**, **T** in case of **warnings** or **errors**, **T** in case of **warnings** or **errors** excluding **style-warnings**.

(*f* **compile-file-pathname** *file* [:output-file *path*] [*other-keyargs*])

▷ Pathname *f* **compile-file** writes to if invoked with the same arguments.

(*f* **load** *path*  $\left\{ \begin{array}{l} \text{:verbose } \text{bool} [\nu \text{*load-verbose*}] \\ \text{:print } \text{bool} [\nu \text{*load-print*}] \\ \text{:if-does-not-exist } \text{bool} [\text{nil}] \\ \text{:external-format } \text{file-format} [\text{default}] \end{array} \right\}$ )

▷ Load source file or compiled file into Lisp environment. Return **T** if successful.

$\nu$ \***compile-file**  $\left\{ \begin{array}{l} \text{pathname*} [\text{NIL}] \\ \text{truename*} [\text{NIL}] \end{array} \right\}$

▷ Input file used by *f* **compile-file**/by *f* **load**.

$\nu$ \***compile**  $\left\{ \begin{array}{l} \text{print*} \\ \text{verbose*} \end{array} \right\}$

▷ Defaults used by *f* **compile-file**/by *f* **load**.

(*f* **parse-namestring** *foo* [*host*]  $\left[ \begin{array}{l} \text{default-pathname} [\nu \text{*default-pathname-defaults*}] \\ \left\{ \begin{array}{l} \text{:start } \text{start} [\text{nil}] \\ \text{:end } \text{end} [\text{NIL}] \\ \text{:junk-allowed } \text{bool} [\text{NIL}] \end{array} \right\} \end{array} \right]$ )

▷ Return pathname converted from string, pathname, or stream *foo*; and position where parsing stopped.

(*f* **merge-pathnames** *path-or-stream*  $\left[ \begin{array}{l} \text{default-path-or-stream} [\nu \text{*default-pathname-defaults*}] \\ \text{default-version} [\text{newest}] \end{array} \right]$ )

▷ Return pathname made by filling in components missing in *path-or-stream* from *default-path-or-stream*.

$\nu$ \***default-pathname-defaults\***

▷ Pathname to use if one is needed and none supplied.

(*f* **user-homedir-pathname** [*host*]) ▷ User's home directory.

(*f* **enough-namestring** *path-or-stream* [*root-path*]  $[\nu \text{*default-pathname-defaults*}]$ )

▷ Return minimal path string that sufficiently describes the path of *path-or-stream* relative to *root-path*.

(*f* **namestring** *path-or-stream*)

(*f* **file-namestring** *path-or-stream*)

(*f* **directory-namestring** *path-or-stream*)

(*f* **host-namestring** *path-or-stream*)

▷ Return string representing full pathname; name, type, and version; directory name; or host name, respectively, of *path-or-stream*.

(*f* **translate-pathname** *path-or-stream* *wildcard-path-a* *wildcard-path-b*)

▷ Translate the path of *path-or-stream* from *wildcard-path-a* into *wildcard-path-b*. Return new path.

(*f* **pathname** *path-or-stream*) ▷ Pathname of *path-or-stream*.

(*f* **logical-pathname** *logical-path-or-stream*)

▷ Logical pathname of *logical-path-or-stream*. Logical pathnames are represented as all-uppercase "[host:];[;]{dir\*<sup>+</sup>};}\*{name\*}\*[{type\*<sup>+</sup>}].[version |\*newest|NEWEST]]".

(*f* **logical-pathname-translations** *logical-host*)

▷ List of (from-wildcard to-wildcard) translations for *logical-host*. **setfable**.

(*f* **load-logical-pathname-translations** *logical-host*)

▷ Load *logical-host*'s translations. Return **NIL** if already loaded; return **T** if successful.

(*f* **translate-logical-pathname** *path-or-stream*)

▷ Physical pathname corresponding to (possibly logical) pathname of *path-or-stream*.

(*f* **probe-file** *file*)

(*f* **truename** *file*)

▷ Canonical name of *file*. If *file* does not exist, return **NIL**/signal **file-error**, respectively.

(*f* **file-write-date** *file*) ▷ Time at which *file* was last written.

(*f* **file-author** *file*) ▷ Return name of *file* owner.

(*f* **file-length** *stream*) ▷ Return length of *stream*.

(*f* **rename-file** *foo* *bar*)

▷ Rename file *foo* to *bar*. Unspecified components of *path bar* default to those of *foo*. Return new pathname, old physical file name, and new physical file name.

(*f* **delete-file** *file*) ▷ Delete *file*. Return **T**.

(*f* **directory** *path*) ▷ List of pathnames matching *path*.

(*f* **ensure-directories-exist** *path* [:verbose *bool*])

▷ Create parts of *path* if necessary. Second return value is **T** if something has been created.

## 14 Packages and Symbols

The Loop Facility provides additional means of symbol handling; see `loop`, page 21.

### 14.1 Predicates

(*f*symbolp *foo*)  
 (*f*packagep *foo*)   ▷ T if *foo* is of indicated type.  
 (*f*keywordp *foo*)

### 14.2 Packages

:*bar*|**keyword**:*bar*   ▷ Keyword, evaluates to :bar.  
*package*:*symbol*   ▷ Exported *symbol* of *package*.  
*package*::*symbol*   ▷ Possibly unexported *symbol* of *package*.

(*m*defpackage *foo* {  
 (:nicknames *nick*\*)\*  
 (:documentation *string*)  
 (:intern *interned-symbol*\*)\*  
 (:use *used-package*\*)\*  
 (:import-from *pkg* *imported-symbol*\*)\*  
 (:shadowing-import-from *pkg* *shd-symbol*\*)\*  
 (:shadow *shd-symbol*\*)\*  
 (:export *exported-symbol*\*)\*  
 (:size *int*)  
 })

▷ Create or modify package *foo* with *interned-symbols*, symbols from *used-packages*, *imported-symbols*, and *shd-symbols*. Add *shd-symbols* to *foo*'s shadowing list.

(*f*make-package *foo* {  
 (:nicknames (*nick*\*)NIL)  
 (:use (*used-package*\*)\*)  
 })

▷ Create package *foo*.

(*f*rename-package *package* *new-name* [*new-nicknames*NIL])  
 ▷ Rename *package*. Return renamed package.

(*m*in-package *foo*)   ▷ Make package *foo* current.

{  
 (*f*use-package)  
 (*f*unuse-package)  
 } *other-packages* [*package*v\*package\*])  
 ▷ Make exported symbols of *other-packages* available in *package*, or remove them from *package*, respectively. Return T.

(*f*package-use-list *package*)

(*f*package-used-by-list *package*)

▷ List of other packages used by/using *package*.

(*f*delete-package *package*)

▷ Delete *package*. Return T if successful.

v\*package\*common-lisp-user   ▷ The current package.

(*f*list-all-packages)   ▷ List of registered packages.

(*f*package-name *package*)   ▷ Name of package.

(*f*package-nicknames *package*)   ▷ Nicknames of package.

(*f*find-package *name*)   ▷ Package with name (case-sensitive).

(*f*find-all-symbols *foo*)

▷ List of symbols *foo* from all registered packages.

{  
 (*f*intern)  
 (*f*find-symbol)  
 } *foo* [*package*v\*package\*])

▷ Intern or find, respectively, symbol *foo* in *package*. Second return value is one of :internal, :external, or :inherited (or NIL if *f*intern has created a fresh symbol).

(*f*unintern *symbol* [*package*v\*package\*])

▷ Remove *symbol* from *package*, return T on success.

{  
 (*f*import)  
 (*f*shadowing-import)  
 } *symbols* [*package*v\*package\*])

▷ Make *symbols* internal to *package*. Return T. In case of a name conflict signal correctable **package-error** or shadow the old symbol, respectively.

(*f*shadow *symbols* [*package*v\*package\*])

▷ Make *symbols* of *package* shadow any otherwise accessible, equally named symbols from other packages. Return T.

(*f*package-shadowing-symbols *package*)

▷ List of symbols of *package* that shadow any otherwise accessible, equally named symbols from other packages.

(*f*export *symbols* [*package*v\*package\*])

▷ Make *symbols* external to *package*. Return T.

(*f*unexport *symbols* [*package*v\*package\*])

▷ Revert *symbols* to internal status. Return T.

{  
 (*m*do-symbols)  
 (*m*do-external-symbols)  
 (*m*do-all-symbols)  
 } (*var* [*package*v\*package\*] [*result*NIL])  
 (*var* [*result*NIL])  
 (declare *decl*\*)\* {  
 (*tag*)  
 (*form*)  
 }\*

▷ Evaluate tagbody-like body with *var* successively bound to every symbol from *package*, to every external symbol from *package*, or to every symbol from all registered packages, respectively. Return values of result. Implicitly, the whole form is a block named NIL.

(*m*with-package-iterator (*foo* *packages* [:internal|:external|:inherited])

(declare *decl*\*)\* *form*<sup>P</sup>)

▷ Return values of forms. In *forms*, successive invocations of (*foo*) return: T if a symbol is returned; a symbol from *packages*; accessibility (:internal, :external, or :inherited); and the package the symbol belongs to.

(*f*require *module* [*paths*NIL])

▷ If not in v\*modules\*, try *paths* to load *module* from. Signal **error** if unsuccessful. Deprecated.

(*f*provide *module*)

▷ If not already there, add *module* to v\*modules\*. Deprecated.

v\*modules\*   ▷ List of names of loaded modules.

### 14.3 Symbols

A symbol has the attributes *name*, home **package**, property list, and optionally value (of global constant or variable *name*) and function (**function**, macro, or special operator *name*).

(*f*make-symbol *name*)

▷ Make fresh, uninterned symbol *name*.

(*f*gensym [*s*0])

▷ Return fresh, uninterned symbol #:sn with *n* from v\*gensym-counter\*. Increment v\*gensym-counter\*.

(*f*gentemp [*prefix*0] [*package*v\*package\*])

▷ Intern fresh symbol in *package*. Deprecated.

(*f*copy-symbol *symbol* [*props*NIL])

▷ Return uninterned copy of *symbol*. If *props* is T, give copy the same value, function and property list.

(*f*symbol-name *symbol*)

(*f*symbol-package *symbol*)

(*f*symbol-plist *symbol*)

(*f*symbol-value *symbol*)

(*f*symbol-function *symbol*)

▷ Name, package, property list, value, or function, respectively, of *symbol*. **setfable**.



## Index

" 33  
 ' 33  
 ( 33  
 () 44  
 ) 33  
 \* 3, 30, 31, 41, 45  
 \*\* 41, 45  
 \*\*\* 45  
 \*BREAK-  
 ON-SIGNALS\* 29  
 \*COMPILE-FILE-  
 PATHNAME\* 44  
 \*COMPILE-FILE-  
 TRUENAME\* 44  
 \*COMPILE-PRINT\* 44  
 \*COMPILE-VERBOSE\*  
 44  
 \*DEBUG-IO\* 40  
 \*DEBUGGER-HOOK\*  
 30  
 \*DEFAULT-  
 PATHNAME-  
 DEFAULTS\* 41  
 \*ERROR-OUTPUT\* 40  
 \*FEATURES\* 34  
 \*GENSYM-COUNTER\*  
 43  
 \*LOAD-PATHNAME\*  
 44  
 \*LOAD-PRINT\* 44  
 \*LOAD-TRUENAME\*  
 44  
 \*LOAD-VERBOSE\* 44  
 \*MACROEXPAND-  
 HOOK\* 45  
 \*MODULES\* 43  
 \*PACKAGE\* 42  
 \*PRINT-ARRAY\* 36  
 \*PRINT-BASE\* 36  
 \*PRINT-CASE\* 36  
 \*PRINT-CIRCLE\* 36  
 \*PRINT-ESCAPE\* 36  
 \*PRINT-GENSYM\* 36  
 \*PRINT-LENGTH\* 36  
 \*PRINT-LEVEL\* 36  
 \*PRINT-LINES\* 36  
 \*PRINT-  
 MISER-WIDTH\* 36  
 \*PRINT-PPRINT-  
 DISPATCH\* 36  
 \*PRINT-PRETTY\* 36  
 \*PRINT-RADIX\* 36  
 \*PRINT-READABLY\*  
 36  
 \*PRINT-  
 RIGHT-MARGIN\* 36  
 \*QUERY-IO\* 40  
 \*RANDOM-STATE\* 4  
 \*READ-BASE\* 33  
 \*READ-DEFAULT-  
 FLOAT-FORMAT\* 33  
 \*READ-EVAL\* 34  
 \*READ-SUPPRESS\* 33  
 \*READTABLE\* 33  
 \*STANDARD-INPUT\*  
 40  
 \*STANDARD-  
 OUTPUT\* 40  
 \*TERMINAL-IO\* 40  
 \*TRACE-OUTPUT\* 46  
 + 3, 26, 45  
 ++ 45  
 +++ 45  
 . 33  
 @ 33  
 - 3, 45  
 : 33  
 / 3, 34, 45  
 // 45  
 /// 45  
 /= 3  
 : 42  
 :: 42  
 :ELEMENT-TYPE 30  
 :ARRAY-HAS-  
 FILL-POINTER-P 10  
 :ARRAY-IN-BOUNDS-P  
 10  
 :ARRAY-RANK 11  
 :ARRAY-RANK-LIMIT 11  
 :ARRAY-ROW-  
 MAJOR-INDEX 11  
 :ARRAY-TOTAL-SIZE 11  
 :ARRAY-TOTAL-  
 SIZE-LIMIT 11  
 :ARRAYP 10  
 :AS 21  
 :ASH 5  
 :+ 34  
 :ASIN 3  
 :ASINH 4  
 :ASSERT 28  
 :ASSOC 9  
 :ASSOC-IF 9  
 :ASSOC-IF-NOT 9  
 :ATAN 3  
 :ATANH 4  
 :#C 34  
 :#O 33  
 :#P 34  
 :#R 33  
 :#S 34  
 :#X 33  
 :## 34  
 :#|# 33  
 &ALLOW-  
 OTHER-KEYS 19

&AUX 19  
 &BODY 19  
 &ENVIRONMENT 19  
 &KEY 19  
 &OPTIONAL 19  
 &REST 19  
 &WHOLE 19  
 ~ ( ~ ) 37  
 ~\* 38  
 ~// 38  
 ~< ~> 38  
 ~< ~> 37  
 ~? 38  
 ~A 37  
 ~B 37  
 ~C 37  
 ~D 37  
 ~E 37  
 ~F 37  
 ~G 37  
 ~I 38  
 ~O 37  
 ~P 37  
 ~R 37  
 ~S 37  
 ~T 38  
 ~W 38  
 ~X 37  
 ~[ ~] 38  
 ~\$ 37  
 ~% 37  
 ~& 37  
 ~^ 38  
 ~\_ 37  
 ~| 37  
 ~{ ~} 38  
 ~~ 37  
 ~↔ 37  
 | 33  
 | | 34  
 | + 3  
 | - 3

ABORT 29  
 ABOVE 21  
 ABS 4  
 ACOS 9  
 ACOS 3  
 ACOSH 4  
 ACROSS 21  
 ADD-METHOD 26  
 ADJOIN 9  
 ADJUST-ARRAY 10  
 ADJUSTABLE-  
 ARRAY-P 10  
 ALLOCATE-INSTANCE  
 25  
 ALPHA-CHAR-P 6  
 ALPHANUMERICP 6  
 ALWAYS 23  
 AND  
 20, 21, 23, 26, 30, 34  
 APPEND 9, 23, 26  
 APPENDING 23  
 APPLY 17  
 APROPOS 45  
 APROPOS-LIST 45  
 AREF 10  
 ARITHMETIC-ERROR  
 31  
 ARITHMETIC-ERROR-  
 OPERANDS 29  
 ARITHMETIC-ERROR-  
 OPERATION 29  
 ARRAY 31  
 ARRAY-DIMENSION 11  
 ARRAY-DIMENSION-  
 LIMIT 11  
 ARRAY-DIMENSIONS  
 11  
 ARRAY-  
 DISPLACEMENT 11  
 ARRAY-  
 ELEMENT-TYPE 30  
 ARRAY-HAS-  
 FILL-POINTER-P 10  
 ARRAY-IN-BOUNDS-P  
 10  
 ARRAY-RANK 11  
 ARRAY-RANK-LIMIT 11  
 ARRAY-ROW-  
 MAJOR-INDEX 11  
 ARRAY-TOTAL-SIZE 11  
 ARRAY-TOTAL-  
 SIZE-LIMIT 11  
 ARRAYP 10  
 AS 21  
 ASH 5  
 ASIN 3  
 ASINH 4  
 ASSERT 28  
 ASSOC 9  
 ASSOC-IF 9  
 ASSOC-IF-NOT 9  
 ATAN 3  
 ATANH 4  
 ATOM 8, 31

BASE-CHAR 31  
 BASE-STRING 31  
 BEING 21  
 BELOW 21  
 BIGNUM 31  
 BIT 11, 31  
 BIT-AND 11

BIT-ANDC1 11  
 BIT-ANDC2 11  
 BIT-EQV 11  
 BIT-IOR 11  
 BIT-NAND 11  
 BIT-NOR 11  
 BIT-NOT 11  
 BIT-ORC1 11  
 BIT-ORC2 11  
 BIT-VECTOR 31  
 BIT-VECTOR-P 10  
 BIT-XOR 11  
 BLOCK 20  
 BOOL 4  
 BOOLE-1 4  
 BOOLE-2 4  
 BOOLE-AND 5  
 BOOLE-ANDC1 5  
 BOOLE-ANDC2 5  
 BOOLE-C1 4  
 BOOLE-C2 4  
 BOOLE-CLR 4  
 BOOLE-EQV 5  
 BOOLE-IOR 5  
 BOOLE-NAND 5  
 BOOLE-NOR 5  
 BOOLE-ORC1 5  
 BOOLE-ORC2 5  
 BOOLE-SET 4  
 BOOLE-XOR 5  
 BOOLEAN 31  
 BOTH-CASE-P 6  
 BOUNDP 15  
 BREAK 46  
 BROADCAST-STREAM  
 31  
 BROADCAST-  
 STREAM-STREAMS  
 39  
 BUILT-IN-CLASS 31  
 BUILT-LAST 9  
 BY 21  
 BYTE 5  
 BYTE-POSITION 5  
 BYTE-SIZE 5

CAAR 8  
 CADR 8  
 CALL-ARGUMENTS-  
 LIMIT 18  
 CALL-METHOD 27  
 CALL-NEXT-METHOD  
 26  
 CAR 8  
 CASE 20  
 CATCH 20  
 CCASE 20  
 CDAR 8  
 CDDR 8  
 CDR 8  
 CEILING 4  
 CELL-ERROR 31  
 CELL-ERROR-NAME 29  
 CERROR 28  
 CHANGE-CLASS 24  
 CHAR 8  
 CHAR-CODE 7  
 CHAR-CODE-LIMIT 7  
 CHAR-DOWNCASE 7  
 CHAR-EQUAL 6  
 CHAR-GREATERP 7  
 CHAR-INT 7  
 CHAR-LESSP 7  
 CHAR-NAME 7  
 CHAR-NOT-EQUAL 6  
 CHAR-NOT-GREATERP  
 7  
 CHAR-NOT-LESSP 7  
 CHAR-UPCASE 7  
 CHAR/= 6  
 CHAR < 6  
 CHAR <= 6  
 CHAR = 6  
 CHAR > 6  
 CHAR >= 6  
 CHARACTER 7, 31, 33  
 CHARACTERP 6  
 CHECK-TYPE 30  
 CIS 4  
 CL 44  
 CL-USER 44  
 CLASS 31  
 CLASS-NAME 24  
 CLASS-OF 24  
 CLEAR-INPUT 39  
 CLEAR-OUTPUT 39  
 CLOSE 40  
 CLR 1  
 CLRHASH 14  
 CODE-CHAR 7  
 COERCE 30  
 COLLECT 23  
 COLLECTING 23  
 COMMON-LISP 44  
 COMMON-LISP-USER  
 44  
 COMPILATION-SPEED  
 46  
 COMPILE 44  
 COMPILE-FILE 44  
 COMPILE-  
 FILE-PATHNAME 44  
 COMPILED-FUNCTION  
 31

COMPILED-  
 FUNCTION-P 44  
 COMPILER-MACRO 44  
 COMPILER-MACRO-  
 FUNCTION 45  
 COMPLEMENT 17  
 COMPLEX 4, 31, 34  
 COMPLEXP 3  
 COMPUTE-  
 APPLICABLE-  
 METHODS 26  
 COMPUTE-RESTARTS  
 29  
 CONCATENATE 12  
 CONCATENATED-  
 STREAM 31  
 CONCATENATED-  
 STREAM-STREAMS  
 39  
 COND 19  
 CONDITION 31  
 CONJUGATE 4  
 CONS 8, 31  
 CONSP 8  
 CONSTANTLY 17  
 CONSTANTP 15  
 CONTINUE 29  
 CONTROL-ERROR 31  
 COPY-ALIST 9  
 COPY-LIST 9  
 COPY-PPRINT-  
 DISPATCH 36  
 COPY-READTABLE 33  
 COPY-SEQ 14  
 COPY-STRUCTURE 15  
 COPY-SYMBOL 43  
 COPY-TREE 10  
 COS 3  
 COSH 3  
 COUNT 12, 23  
 COUNT-IF 12  
 COUNT-IF-NOT 12  
 COUNTING 23  
 CTYPESCASE 30

DEBUG 46  
 DECF 3  
 DECLAIM 46  
 DECLARATION 46  
 DECLARE 46  
 DECODE-FLOAT 6  
 DECODE-UNIVERSAL-  
 TIME 47  
 DEFCCLASS 24  
 DEFCONSTANT 16  
 DEFGENERIC 25  
 DEFINE-COMPILER-  
 MACRO 18  
 DEFINE-CONDITION 27  
 DEFINE-METHOD-  
 COMBINATION 26, 27  
 DEFINE-  
 MODIFY-MACRO 19  
 DEFINE-  
 SETF-EXPANDER 19  
 DEFINE-  
 SYMBOL-MACRO 18  
 DEFMACRO 18  
 DEFMETHOD 25  
 DEFPACKAGE 42  
 DEFPARAMETER 16  
 DEFSETF 18  
 DEFSTRUCT 15  
 DEFTYPE 30  
 DEFUN 17  
 DEFVAR 16  
 DELETE 13  
 DELETE-DUPLICATES  
 13  
 DELETE-FILE 41  
 DELETE-IF 13  
 DELETE-IF-NOT 13  
 DELETE-PACKAGE 42  
 DENOMINATOR 4  
 DEPOSIT-FIELD 5  
 DESCRIBE 46  
 DESCRIBE-OBJECT 46  
 DESTRUCTURING-  
 BIND 17  
 DIGIT-CHAR 7  
 DIGIT-CHAR-P 6  
 DIRECTORY 41  
 DIRECTORY-  
 NAMESTRING 41  
 DISASSEMBLE 46  
 DIVISION-BY-ZERO 31  
 DO 21, 23  
 DO-ALL-SYMBOLS 43  
 DO-EXTERNAL-  
 SYMBOLS 43  
 DO-SYMBOLS 43  
 DO\* 21  
 DOCUMENTATION 44  
 DOING 23  
 DOLIST 21  
 DOTIMES 21  
 DOUBLE-FLOAT 31, 34  
 DOUBLE-  
 FLOAT-EPSILON 6  
 DOUBLE-FLOAT-  
 NEGATIVE-EPSILON  
 6  
 DOWNFROM 21  
 DOWNTO 21  
 DPB 5

$$(\text{eval-when} (\left\{ \left\{ \begin{array}{l} \text{:compile-toplevel} \\ \text{:load-toplevel} \\ \text{:execute} \end{array} \right\} \mid \text{compile} \right\} \right\}) \text{form}^{\text{P}})$$

▷ Return values of forms if eval-when is in the top-level of a file being compiled, in the top-level of a compiled file being loaded, or anywhere, respectively. Return NIL if forms are not evaluated. (compile, load and eval deprecated.)

$$(\text{locally} (\widehat{\text{declare}} \text{decl}^*) \text{form}^{\text{P}})$$

▷ Evaluate forms in a lexical environment with declarations decl in effect. Return values of forms.

$$(\text{mwith-compilation-unit} ([\text{override} \text{bool}^{\text{N}}]) \text{form}^{\text{P}})$$

▷ Return values of forms. Warnings deferred by the compiler until end of compilation are deferred until the end of evaluation of forms.

$$(\text{sload-time-value} \text{form} [\widehat{\text{read-only}}^{\text{N}}])$$

▷ Evaluate form at compile time and treat its value as literal at run time.

$$(\text{squote} \widehat{\text{foo}}) \quad \triangleright \text{Return unevaluated } \text{foo}.$$

$$(\text{gmake-load-form} \text{foo} [\text{environment}])$$

▷ Its methods are to return a creation form which on evaluation at load time returns an object equivalent to foo, and an optional initialization form which on evaluation performs some initialization of the object.

$$(\text{fmake-load-form-saving-slots} \text{foo} \left\{ \begin{array}{l} \text{:slot-names} \text{slots}^{\text{all local slots}} \\ \text{:environment} \text{environment} \end{array} \right\})$$

▷ Return a creation form and an initialization form which on evaluation construct an object equivalent to foo with slots initialized with the corresponding values from foo.

$$(\text{fmacro-function} \text{symbol} [\text{environment}])$$

$$(\text{fcompiler-macro-function} \left\{ \begin{array}{l} \text{name} \\ \text{:setf name} \end{array} \right\} [\text{environment}])$$

▷ Return specified macro function, or compiler macro function, respectively, if any. Return NIL otherwise. setfable.

$$(\text{f eval} \text{arg})$$

▷ Return values of value of arg evaluated in global environment.

## 15.3 REPL and Debugging

$$\vee + \mid \vee + + \mid \vee + + +$$

$$\vee * \mid \vee ** \mid \vee ***$$

$$\vee / \mid \vee // \mid \vee ///$$

▷ Last, penultimate, or antepenultimate form evaluated in the REPL, or their respective primary value, or a list of their respective values.

$$\vee \text{ } \quad \triangleright \text{Form currently being evaluated by the REPL.}$$

$$(\text{f apropos} \text{string} [\text{package}^{\text{N}}])$$

▷ Print interned symbols containing string.

$$(\text{f apropos-list} \text{string} [\text{package}^{\text{N}}])$$

▷ List of interned symbols containing string.

$$(\text{f dribble} [\text{path}])$$

▷ Save a record of interactive session to file at path. Without path, close that file.

$$(\text{f ed} [\text{file-or-function}^{\text{N}}])$$

▷ Invoke editor if possible.

$$\left\{ \begin{array}{l} \text{:fmacroexpand-1} \\ \text{:fmacroexpand} \end{array} \right\} \text{form} [\text{environment}^{\text{N}}]$$

▷ Return macro expansion, once or entirely, respectively, of form and T if form was a macro form. Return form and NIL otherwise.

$$\vee * \text{macroexpand-hook}^*$$

▷ Function of arguments expansion function, macro form, and environment called by fmacroexpand-1 to generate macro expansions.



- (*m*trace {function  
(setf function)}\*)  
 ▷ Cause *functions* to be traced. With no arguments, return list of traced functions.
- (*m*untrace {function  
(setf function)}\*)  
 ▷ Stop *functions*, or each currently traced function, from being traced.
- v*\*trace-output\*  
 ▷ Output stream *m*trace and *m*time send their output to.
- (*m*step form)  
 ▷ Step through evaluation of *form*. Return values of form.
- (*f*break [control arg\*])  
 ▷ Jump directly into debugger; return NIL. See page 36, *f*format, for *control* and *args*.
- (*m*time form)  
 ▷ Evaluate *forms* and print timing information to *v*\*trace-output\*. Return values of form.
- (*f*inspect foo)   ▷ Interactively give information about *foo*.
- (*f*describe foo [*stream* *v*\*standard-output\*])  
 ▷ Send information about *foo* to *stream*.
- (*g*describe-object foo [*stream*])  
 ▷ Send information about *foo* to *stream*. Called by *f*describe.
- (*f*disassemble function)  
 ▷ Send disassembled representation of *function* to *v*\*standard-output\*. Return NIL.
- (*f*room [{NIL|default|T}|*default*])  
 ▷ Print information about internal storage management to *\*standard-output\**.

## 15.4 Declarations

- (*f*proclaim decl)  
 (*m*declaim decl\*)  
 ▷ Globally make declaration(s) *decl*. *decl* can be: **declaration**, **type**, **ftype**, **inline**, **notinline**, **optimize**, or **special**. See below.
- (declare decl\*)  
 ▷ Inside certain forms, locally make declarations *decl*\*. *decl* can be: **dynamic-extent**, **type**, **ftype**, **ignorable**, **ignore**, **inline**, **notinline**, **optimize**, or **special**. See below.
- (declaration foo\*)  
 ▷ Make *foos* names of declarations.
- (dynamic-extent variable\* (function function)\*)  
 ▷ Declare lifetime of *variables* and/or *functions* to end when control leaves enclosing block.
- ([type] type variable\*)  
 (ftype type function\*)  
 ▷ Declare *variables* or *functions* to be of *type*.
- {ignorable} {var  
{ignore} {(function function)}}\*)  
 ▷ Suppress warnings about used/unused bindings.
- (inline function\*)  
 (notinline function\*)  
 ▷ Tell compiler to integrate/not to integrate, respectively, called *functions* into the calling routine.
- (optimize {compilation-speed|(compilation-speed *n*<sub>☐</sub>)  
 debug|(debug *n*<sub>☐</sub>)  
 safety|(safety *n*<sub>☐</sub>)  
 space|(space *n*<sub>☐</sub>)  
 speed|(speed *n*<sub>☐</sub>)
- ▷ Tell compiler how to optimize.
- n*
- = 0 means unimportant,
- n*
- = 1 is neutral,
- n*
- = 3 means important.
- (special var\*)   ▷ Declare *vars* to be dynamic.

## 16 External Environment

- (*f*get-internal-real-time)  
 (*f*get-internal-run-time)  
 ▷ Current time, or computing time, respectively, in clock ticks.
- c*internal-time-units-per-second  
 ▷ Number of clock ticks per second.
- (*f*encode-universal-time sec min hour date month year [zone<sub>☐</sub>])  
 (*f*get-universal-time)  
 ▷ Seconds from 1900-01-01, 00:00, ignoring leap seconds.
- (*f*decode-universal-time universal-time [time-zone<sub>☐</sub>])  
 (*f*get-decoded-time)  
 ▷ Return second, minute, hour, date, month, year, day, daylight-p, and zone.
- (*f*short-site-name)  
 (*f*long-site-name)  
 ▷ String representing physical location of computer.
- {*f*lisp-implementation}  
 {*f*software} - {type  
{*f*machine}    {version}}
- ▷
- Name
- or
- version
- of implementation, operating system, or hardware, respectively.
- (*f*machine-instance)   ▷ Computer name.

DRIBBLE 45  
 DYNAMIC-EXTENT 46  
  
 EACH 21  
 ECASE 20  
 ECHO-STREAM 31  
 ECHO-STREAM-  
 INPUT-STREAM 39  
 ECHO-STREAM-  
 OUTPUT-STREAM  
 39  
 ED 45  
 EIGHTH 8  
 ELSE 23  
 ELT 12  
 ENCODE-UNIVERSAL-  
 TIME 47  
 END 23  
 END-OF-FILE 31  
 ENDP 8  
 ENOUGH-  
 NAMESTRING 41  
 ENSURE-  
 DIRECTORIES-EXIST  
 41  
 ENSURE-GENERIC-  
 FUNCTION 25  
 EQ 15  
 EQL 15, 30  
 EQUAL 15  
 EQUALP 15  
 ERROR 28, 31  
 ETYPCASE 30  
 EVAL 45  
 EVAL-WHEN 45  
 EVENP 3  
 EVERY 12  
 EXP 3  
 EXPORT 43  
 EXPT 3  
 EXTENDED-CHAR 31  
 EXTERNAL-SYMBOL  
 23  
 EXTERNAL-SYMBOLS  
 23  
  
 FBOUNDP 16  
 FCEILING 4  
 FDEFINITION 18  
 FFLOOR 4  
 FIFTH 8  
 FILE-AUTHOR 41  
 FILE-ERROR 31  
 FILE-ERROR-  
 PATHNAME 29  
 FILE-LENGTH 41  
 FILE-NAMESTRING 41  
 FILE-POSITION 39  
 FILE-STREAM 31  
 FILE-STREAM-LENGTH  
 39  
 FILE-WRITE-DATE 41  
 FILL 12  
 FILL-POINTER 11  
 FINALLY 23  
 FIND 13  
 FIND-ALL-SYMBOLS 42  
 FIND-CLASS 24  
 FIND-IF 13  
 FIND-IF-NOT 13  
 FIND-METHOD 26  
 FIND-PACKAGE 42  
 FIND-RESTART 29  
 FIND-SYMBOL 42  
 FINISH-OUTPUT 39  
 FIRST 8  
 FIXNUM 31  
 FLET 17  
 FLOAT 4, 31  
 FLOAT-DIGITS 6  
 FLOAT-PRECISION 6  
 FLOAT-RADIX 6  
 FLOAT-SIGN 4  
 FLOATING-  
 POINT-INEXACT 31  
 FLOATING-  
 POINT-INVALID-  
 OPERATION 31  
 FLOATING-POINT-  
 OVERFLOW 31  
 FLOATING-POINT-  
 UNDERFLOW 31  
 FLOATP 3  
 FLOOR 4  
 FMAKUNBOUND 18  
 FOR 21  
 FORCE-OUTPUT 39  
 FORMAT 36  
 FORMATTER 36  
 FOURTH 8  
 FRESH-LINE 34  
 FROM 21  
 FROUND 4  
 FTRUNCATE 4  
 FTYPE 46  
 FUNCALL 17  
 FUNCTION  
 17, 31, 34, 44  
 FUNCTION-  
 KEYWORDS 26  
 FUNCTION-LAMBDA-  
 EXPRESSION 18  
 FUNCTIONP 16  
  
 GCD 3  
 GENERIC-FUNCTION  
 31  
 GENSYM 43  
  
 GENTEMP 43  
 GET 16  
 GET-DECODED-TIME  
 47  
 GET-  
 DISPATCH-MACRO-  
 CHARACTER 33  
 GET-INTERNAL-  
 REAL-TIME 47  
 GET-INTERNAL-  
 RUN-TIME 47  
 GET-MACRO-  
 CHARACTER 33  
 GET-OUTPUT-  
 STREAM-STRING 39  
 GET-PROPERTIES 16  
 GET-SETF-EXPANSION  
 19  
 GET-UNIVERSAL-TIME  
 47  
 GETF 16  
 GETHASH 14  
 GO 20  
 GRAPHIC-CHAR-P 6  
  
 HANDLER-BIND 28  
 HANDLER-CASE 28  
 HASH-KEY 21, 23  
 HASH-KEYS 21  
 HASH-TABLE 31  
 HASH-TABLE-COUNT  
 14  
 HASH-TABLE-P 14  
 HASH-TABLE-  
 REHASH-SIZE 14  
 HASH-  
 TABLE-REHASH-  
 THRESHOLD 14  
 HASH-TABLE-SIZE 14  
 HASH-TABLE-TEST 14  
 HASH-VALUE 21, 23  
 HASH-VALUES 23  
 HOST-NAMESTRING 41  
  
 IDENTITY 17  
 IF 19, 23  
 IGNORABLE 46  
 IGNORE 46  
 IGNORE-ERRORS 28  
 IMAGPART 4  
 IMPORT 43  
 IN 21, 23  
 IN-PACKAGE 42  
 INCF 3  
 INITIALIZE-INSTANCE  
 24  
 INITIALLY 23  
 INLINE 46  
 INPUT-STREAM-P 32  
 INSPECT 46  
 INTEGER 31  
 INTEGER-  
 DECODE-FLOAT 6  
 INTEGER-LENGTH 5  
 INTEGERP 3  
 INTERACTIVE-  
 STREAM-P 32  
 INTERN 42  
 INTERNAL-TIME-  
 UNITS-PER-SECOND  
 47  
 INTERSECTION 10  
 INTO 23  
 INVALID-  
 METHOD-ERROR 26  
 INVOKE-DEBUGGER 28  
 INVOKE-RESTART 29  
 INVOKE-RESTART-  
 INTERACTIVELY 29  
 ISQRT 3  
 IT 23  
  
 KEYWORD 31, 42, 44  
 KEYWORDP 42  
  
 LABELS 17  
 LAMBDA 17  
 LAMBDA-  
 LIST-KEYWORDS 19  
 LAMBDA-  
 PARAMETERS-LIMIT  
 18  
 LAST 8  
 LCM 3  
 LDB 5  
 LDB-TEST 5  
 LDIFF 9  
 LEAST-NEGATIVE-  
 DOUBLE-FLOAT 6  
 LEAST-NEGATIVE-  
 LONG-FLOAT 6  
 LEAST-NEGATIVE-  
 NORMALIZED-  
 DOUBLE-FLOAT 6  
 LEAST-NEGATIVE-  
 NORMALIZED-  
 LONG-FLOAT 6  
 LEAST-NEGATIVE-  
 NORMALIZED-  
 SHORT-FLOAT 6  
 LEAST-NEGATIVE-  
 NORMALIZED-  
 SINGLE-FLOAT 6  
 LEAST-NEGATIVE-  
 SHORT-FLOAT 6  
 LEAST-NEGATIVE-  
 SINGLE-FLOAT 6  
  
 LEAST-POSITIVE-  
 DOUBLE-FLOAT 6  
 LEAST-POSITIVE-  
 LONG-FLOAT 6  
 LEAST-POSITIVE-  
 NORMALIZED-  
 DOUBLE-FLOAT 6  
 LEAST-POSITIVE-  
 NORMALIZED-  
 LONG-FLOAT 6  
 LEAST-POSITIVE-  
 NORMALIZED-  
 SHORT-FLOAT 6  
 LEAST-POSITIVE-  
 NORMALIZED-  
 SINGLE-FLOAT 6  
 LEAST-POSITIVE-  
 SHORT-FLOAT 6  
 LEAST-POSITIVE-  
 SINGLE-FLOAT 6  
 LENGTH 12  
 LET 16  
 LET\* 16  
 LISP-  
 IMPLEMENTATION-  
 TYPE 47  
 LISP-  
 IMPLEMENTATION-  
 VERSION 47  
 LIST 8, 26, 31  
 LIST-ALL-PACKAGES  
 42  
 LIST-LENGTH 8  
 LIST\* 8  
 LISTEN 39  
 LISTP 8  
 LOAD 44  
 LOAD-LOGICAL-  
 PATHNAME-  
 TRANSLATIONS 41  
 LOAD-TIME-VALUE 45  
 LOCALLY 45  
 LOG 3  
 LOGAND 5  
 LOGANDC1 5  
 LOGANDC2 5  
 LOGBITP 5  
 LOGCOUNT 5  
 LOGEQV 5  
 LOGICAL-PATHNAME  
 31, 41  
 LOGICAL-PATHNAME-  
 TRANSLATIONS 41  
 LOGIOR 5  
 LOGNAND 5  
 LOGNOR 5  
 LOGNOT 5  
 LOGORC1 5  
 LOGORC2 5  
 LOGTEST 5  
 LOGXOR 5  
 LONG-FLOAT 31, 34  
 LONG-FLOAT-EPSILON  
 6  
 LONG-FLOAT-  
 NEGATIVE-EPSILON  
 6  
 LONG-SITE-NAME 47  
 LOOP 21  
 LOOP-FINISH 23  
 LOWER-CASE-P 6  
  
 MACHINE-INSTANCE  
 47  
 MACHINE-TYPE 47  
 MACHINE-VERSION 47  
 MACRO-FUNCTION 45  
 MACROEXPAND 45  
 MACROEXPAND-1 45  
 MACROLET 18  
 MAKE-ARRAY 10  
 MAKE-BROADCAST-  
 STREAM 39  
 MAKE-  
 CONCATENATED-  
 STREAM 39  
 MAKE-CONDITION 28  
 MAKE-  
 DISPATCH-MACRO-  
 CHARACTER 33  
 MAKE-ECHO-STREAM  
 39  
 MAKE-HASH-TABLE 14  
 MAKE-INSTANCE 24  
 MAKE-INSTANCES-  
 OBSOLETE 24  
 MAKE-LIST 8  
 MAKE-LOAD-FORM 45  
 MAKE-LOAD-FORM-  
 SAVING-SLOTS 45  
 MAKE-METHOD 27  
 MAKE-PACKAGE 42  
 MAKE-PATHNAME 40  
 MAKE-  
 RANDOM-STATE 4  
 MAKE-SEQUENCE 12  
 MAKE-STRING 7  
 MAKE-STRING-  
 INPUT-STREAM 39  
 MAKE-STRING-  
 OUTPUT-STREAM  
 39  
 MAKE-SYMBOL 43  
 MAKE-SYNONYM-  
 STREAM 39  
 MAKE-TWO-  
 WAY-STREAM 39  
 MAKUNBOUND 16  
 MAP 14  
  
 MAP-INTO 14  
 MAPC 9  
 MAPCAN 9  
 MAPCAR 9  
 MAPCON 9  
 MAPHASH 14  
 MAPL 9  
 MAPLIST 9  
 MASK-FIELD 5  
 MAX 4, 26  
 MAXIMIZE 23  
 MAXIMIZING 23  
 MEMBER 8, 30  
 MEMBER-IF 8  
 MEMBER-IF-NOT 8  
 MERGE 12  
 MERGE-PATHNAMES  
 41  
 METHOD 31  
 METHOD-  
 COMBINATION 31, 44  
 METHOD-  
 COMBINATION-  
 ERROR 26  
 METHOD-QUALIFIERS  
 26  
 MIN 4, 26  
 MINIMIZE 23  
 MINIMIZING 23  
 MINUSP 3  
 MISMATCH 12  
 MOD 4, 30  
 MOST-NEGATIVE-  
 DOUBLE-FLOAT 6  
 MOST-NEGATIVE-  
 SINGLE-FLOAT 6  
 MOST-NEGATIVE-  
 FIXNUM 6  
 MOST-NEGATIVE-  
 LONG-FLOAT 6  
 MOST-NEGATIVE-  
 SHORT-FLOAT 6  
 MOST-NEGATIVE-  
 SINGLE-FLOAT 6  
 MOST-POSITIVE-  
 DOUBLE-FLOAT 6  
 MOST-POSITIVE-  
 FIXNUM 6  
 MOST-POSITIVE-  
 LONG-FLOAT 6  
 MOST-POSITIVE-  
 SHORT-FLOAT 6  
 MOST-POSITIVE-  
 SINGLE-FLOAT 6  
 MUFFLE-WARNING 29  
 MULTIPLE-  
 VALUE-BIND 16  
 MULTIPLE-  
 VALUE-CALL 17  
 MULTIPLE-  
 VALUE-LIST 17  
 MULTIPLE-  
 VALUE-PROG1 20  
 MULTIPLE-  
 VALUE-SETQ 16  
 MULTIPLE-  
 VALUES-LIMIT 18  
  
 NAME-CHAR 7  
 NAMED 21  
 NAMESTRING 41  
 NBTLAST 9  
 NCONC 9, 23, 26  
 NCONCING 23  
 NEVER 23  
 NEWLINE 6  
 NEXT-METHOD-P 25  
 NIL 2, 44  
 NINTERSECTION 10  
 NINTH 8  
 NO-APPLICABLE-  
 METHOD 26  
 NO-NEXT-METHOD 26  
 NOT 15, 30, 34  
 NOTANY 12  
 NOTEVERY 12  
 NOTINLINE 46  
 NRECONC 9  
 NREVERSE 12  
 NSET-DIFFERENCE 10  
 NSET-EXCLUSIVE-OR  
 10  
 NSTRING-CAPITALIZE  
 7  
 NSTRING-DOWNCASE  
 7  
 NSTRING-UPCASE 7  
 NSUBLIS 10  
 NSUBST 10  
 NSUBST-IF 10  
 NSUBST-IF-NOT 10  
 NSUBSTITUTE 13  
 NSUBSTITUTE-IF 13  
 NSUBSTITUTE-IF-NOT  
 13  
 NTH 8  
 NTH-VALUE 17  
 NTHCDR 8  
 NULL 8, 31  
 NUMBER 31  
 NUMBERP 3  
 NUMERATOR 4  
 NUNION 10  
  
 ODDP 3  
 OF 21, 23  
 OF-TYPE 21  
 ON 21  
 OPEN 39  
 OPEN-STREAM-P 32



OPTIMIZE 46  
OR 20, 26, 30, 34  
OTHERWISE 20, 30  
OUTPUT-STREAM-P 32

PACKAGE 31  
PACKAGE-ERROR 31  
PACKAGE-ERROR-PACKAGE 29  
PACKAGE-NAME 42  
PACKAGE-NICKNAMES 42  
PACKAGE-SHADOWING-SYMBOLS 43  
PACKAGE-USE-LIST 42  
PACKAGE-USED-BY-LIST 42  
PACKAGEP 42  
PAIRLIS 9  
PARSE-ERROR 31  
PARSE-INTEGER 8  
PARSE-NAMESTRING 41  
PATHNAME 31, 41  
PATHNAME-DEVICE 40  
PATHNAME-DIRECTORY 40  
PATHNAME-HOST 40  
PATHNAME-MATCH-P 32  
PATHNAME-NAME 40  
PATHNAME-TYPE 40  
PATHNAME-VERSION 40  
PATHNAMEP 32  
PEEK-CHAR 32  
PHASE 4  
PI 3  
PLUSP 3  
POP 9  
POSITION 13  
POSITION-IF 13  
POSITION-IF-NOT 13  
PPRINT 34  
PPRINT-DISPATCH 36  
PPRINT-EXIT-IF-LIST-EXHAUSTED 35  
PPRINT-FILL 35  
PPRINT-INDENT 35  
PPRINT-LINEAR 35  
PPRINT-LOGICAL-BLOCK 35  
PPRINT-NEWLINE 36  
PPRINT-POP 35  
PPRINT-TAB 35  
PPRINT-TABULAR 35  
PRESENT-SYMBOL 23  
PRESENT-SYMBOLS 23  
PRIN1 34  
PRIN1-TO-STRING 34  
PRINC 34  
PRINC-TO-STRING 34  
PRINT 34  
PRINT-NOT-READABLE 31  
PRINT-NOT-READABLE-OBJECT 29  
PRINT-OBJECT 34  
PRINT-UNREADABLE-OBJECT 34  
PROBE-FILE 41  
PROCLAIM 46  
PROG 20  
PROG1 20  
PROG2 20  
PROG\* 20  
PROGN 20, 26  
PROGRAM-ERROR 31  
PROGV 16  
PROVIDE 43  
PSETF 16  
PSETQ 16  
PUSH 9  
PUSHNEW 9

QUOTE 33, 45

RANDOM 4  
RANDOM-STATE 31  
RANDOM-STATE-P 3  
RASSOC 9  
RASSOC-IF 9  
RASSOC-IF-NOT 9  
RATIO 31, 34  
RATIONAL 4, 31  
RATIONALIZE 4  
RATIONALP 3  
READ 32  
READ-BYTE 32  
READ-CHAR 32  
READ-CHAR-NO-HANG 32  
READ-DELIMITED-LIST 32  
READ-FROM-STRING 32  
READ-LINE 32  
READ-PRESERVING-WHITESPACE 32

READ-SEQUENCE 33  
READER-ERROR 31  
READTABLE 31  
READTABLE-CASE 33  
READTABLEP 32  
REAL 31  
REALP 3  
REALPART 4  
REDUCE 14  
REINITIALIZE-INSTANCE 24  
REM 4  
REMF 16  
REMHASH 14  
REMOVE 13  
REMOVE-DUPLICATES 13  
REMOVE-IF 13  
REMOVE-IF-NOT 13  
REMOVE-METHOD 26  
REMPROP 16  
RENAME-FILE 41  
RENAME-PACKAGE 42  
REPEAT 23  
REPLACE 13  
REQUIRE 43  
REST 8  
RESTART 31  
RESTART-BIND 29  
RESTART-CASE 28  
RESTART-NAME 29  
RETURN 20, 23  
RETURN-FROM 20  
REVAPPEND 9  
REVERSE 12  
ROOM 46  
ROTATEF 16  
ROUND 4  
ROW-MAJOR-AREF 10  
RPLACA 9  
RPLACD 9

SAFETY 46  
SATISFIES 30  
SBIT 11  
SCALE-FLOAT 6  
SCHAR 8  
SEARCH 13  
SECOND 8  
SEQUENCE 31  
SERIOUS-CONDITION 31  
SET 16  
SET-DIFFERENCE 10  
SET-DISPATCH-MACRO-CHARACTER 33  
SET-EXCLUSIVE-OR 10  
SET-MACRO-CHARACTER 33  
SET-PPRINT-DISPATCH 36  
SET-SYNTAX-FROM-CHAR 33  
SETF 16, 44  
SETQ 16  
SEVENTH 8  
SHADOW 43  
SHADOWING-IMPORT 43  
SHARED-INITIALIZE 25  
SHIFTF 16  
SHORT-FLOAT 31, 34  
SHORT-FLOAT-EPSILON 6  
SHORT-FLOAT-NEGATIVE-EPSILON 6  
SHORT-SITE-NAME 47  
SIGNAL 28  
SIGNED-BYTE 31  
SIGNUM 4  
SIMPLE-ARRAY 31  
SIMPLE-BASE-STRING 31  
SIMPLE-BIT-VECTOR 31  
SIMPLE-BIT-VECTOR-P 10  
SIMPLE-CONDITION 31  
SIMPLE-CONDITION-FORMAT-ARGUMENTS 29  
SIMPLE-CONDITION-FORMAT-CONTROL 29  
SIMPLE-ERROR 31  
SIMPLE-STRING 31  
SIMPLE-STRING-P 7  
SIMPLE-TYPE-ERROR 31  
SIMPLE-VECTOR 31  
SIMPLE-VECTOR-P 10  
SIMPLE-WARNING 31  
SIN 3  
SINGLE-FLOAT 31, 34  
SINGLE-FLOAT-EPSILON 6  
SINGLE-FLOAT-NEGATIVE-EPSILON 6  
SINH 3  
SIXTH 8

SLEEP 20  
SLOT-BOUND 24  
SLOT-EXISTS-P 23  
SLOT-MAKUNBOUND 24  
SLOT-MISSING 25  
SLOT-UNBOUND 25  
SLOT-VALUE 24  
SOFTWARE-TYPE 47  
SOFTWARE-VERSION 47  
SOME 12  
SORT 12  
SPACE 6, 46  
SPECIAL 46  
SPECIAL-OPERATOR-P 44  
SPEED 46  
SQRT 3  
STABLE-SORT 12  
STANDARD 26  
STANDARD-CHAR 6, 31  
STANDARD-CHAR-P 6  
STANDARD-CLASS 31  
STANDARD-GENERIC-FUNCTION 31  
STANDARD-METHOD 31  
STANDARD-OBJECT 31  
STEP 46  
STORAGE-CONDITION 31  
STORE-VALUE 29  
STREAM 31  
STREAM-ELEMENT-TYPE 30  
STREAM-ERROR 31  
STREAM-ERROR-STREAM 29  
STREAM-EXTERNAL-FORMAT 40  
STREAMP 32  
STRING 7, 31  
STRING-CAPITALIZE 7  
STRING-DOWNCASE 7  
STRING-EQUAL 7  
STRING-GREATERP 7  
STRING-LEFT-TRIM 7  
STRING-LESSP 7  
STRING-NOT-EQUAL 7  
STRING-NOT-GREATERP 7  
STRING-NOT-LESSP 7  
STRING-RIGHT-TRIM 7  
STRING-STREAM 31  
STRING-TRIM 7  
STRING-UPCASE 7  
STRING/= 7  
STRING< 7  
STRING<= 7  
STRING= 7  
STRING> 7  
STRING>= 7  
STRINGP 7  
STRUCTURE 44  
STRUCTURE-CLASS 31  
STRUCTURE-OBJECT 31  
STYLE-WARNING 31  
SUBLIS 10  
SUBSEQ 12  
SUBSETP 8  
SUBST 10  
SUBST-IF 10  
SUBST-IF-NOT 10  
SUBSTITUTE 13  
SUBSTITUTE-IF 13  
SUBSTITUTE-IF-NOT 13  
SUBTYPEP 30  
SUM 23  
SUMMING 23  
SVREF 11  
SXHASH 14  
SYMBOL 23, 31, 43  
SYMBOL-FUNCTION 43  
SYMBOL-MACROLET 18  
SYMBOL-NAME 43  
SYMBOL-PACKAGE 43  
SYMBOL-PLIST 43  
SYMBOL-VALUE 43  
SYMBOLP 42  
SYMBOLS 23  
SYNONYM-STREAM 31  
SYNONYM-STREAM-SYMBOL 39

T 2, 31, 44  
TAGBODY 20  
TAILP 8  
TAN 3  
TANH 3  
TENTH 8  
TERPRI 34  
THE 21, 30  
THEN 21  
THEREIS 23  
THIRD 8  
THROW 20  
TIME 46  
TO 21  
TRACE 46

TRANSLATE-LOGICAL-PATHNAME 41  
TRANSLATE-PATHNAME 41  
TREE-EQUAL 10  
TRUENAME 41  
TRUNCATE 4  
TWO-WAY-STREAM 31  
TWO-WAY-STREAM-INPUT-STREAM 39  
TWO-WAY-STREAM-OUTPUT-STREAM 39  
TYPE 44, 46  
TYPE-ERROR 31  
TYPE-ERROR-DATUM 29  
TYPE-ERROR-EXPECTED-TYPE 29  
TYPE-OF 30  
TYPECASE 30  
TYPEP 30

UNBOUND-SLOT 31  
UNBOUND-SLOT-INSTANCE 29  
UNBOUND-VARIABLE 31  
UNDEFINED-FUNCTION 31  
UNEXPORT 43  
UNINTERN 42  
UNION 10  
UNLESS 19, 23  
UNREAD-CHAR 32  
UNSIGNED-BYTE 31  
UNTIL 23  
UNTRACE 46  
UNUSE-PACKAGE 42  
UNWIND-PROTECT 20  
UPDATE-INSTANCE-FOR-DIFFERENT-CLASS 24  
UPDATE-INSTANCE-FOR-REDEFINED-CLASS 24  
UPFPROM 21  
UPGRADED-ARRAY-ELEMENT-TYPE 30  
UPGRADED-COMPLEX-PART-TYPE 6  
UPPER-CASE-P 6  
UPTO 21  
USE-PACKAGE 42  
USE-VALUE 29  
USER-HOMEDIR-PATHNAME 41  
USING 21, 23

V 38  
VALUES 17, 30  
VALUES-LIST 17  
VARIABLE 44  
VECTOR 11, 31  
VECTOR-POP 11  
VECTOR-PUSH 11  
VECTOR-EXTEND 11  
VECTORP 10

WARN 28  
WARNING 31  
WHEN 19, 23  
WHILE 23  
WILD-PATHNAME-P 32  
WITH 21  
WITH-ACCESSORS 24  
WITH-COMPILE-UNIT 45  
WITH-CONDITION-RESTARTS 29  
WITH-HASH-TABLE-ITERATOR 14  
WITH-INPUT-FROM-STRING 40  
WITH-OPEN-FILE 40  
WITH-OPEN-STREAM 40  
WITH-OUTPUT-TO-STRING 40  
WITH-PACKAGE-ITERATOR 43  
WITH-SIMPLE-RESTART 28  
WITH-SLOTS 24  
WITH-STANDARD-IO-SYNTAX 32  
WRITE 35  
WRITE-BYTE 35  
WRITE-CHAR 35  
WRITE-LINE 35  
WRITE-SEQUENCE 35  
WRITE-STRING 35  
WRITE-TO-STRING 35

Y-OR-N-P 32  
YES-OR-NO-P 32

ZEROP 3